Rechenzeitvergleich serieller und paralleler Berechnungen unter Verwendung von C/C++ Bibliotheken NVIDIA CUDA® am Beispiel der Tourenplanung

Gábor Bárdos

(Matrikelnummer: 70410279)

Eingereichte Abschlussarbeit

zur Erlangung des Grades

Bachelor of Science

im Studiengang

Logistik und Informationsmanagement

an der Karl-Scharfenberg-Fakultät

der Ostfalia Hochschule für angewandte Wissenschaften

Erster Prüfer: Prof. Dr. Ronny Hansmann Eingereicht am: 12.03.2018

Zweiter Prüfer: Dipl.-Ing. (FH), M.Sc.Eng. Marko Apel

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides Statt, dass ich die vorliegende Arbeit selbstständig und ohne unerlaubte Hilfe angefertigt, andere als die angegebenen Quellen nicht benutzt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Weyhausen, 12.03.2018

(Unterschrift)

Inhaltsverzeichnis

1. Einleitung

Dieses Kapitel stellt die Einleitung der Bachelorthesis dar. Es werden Motivation, Aufgabenstellung sowie Zielsetzung erläutert.

1.1 Ausgangssituation und Motivation

Im Zuge der schnell voranschreitenden Digitalisierung erhalten Microchips eine immer wichtigere Bedeutung. Sie befinden sich heutzutage in nahezu allen elektronischen Geräten und sind aus der Industrie nicht mehr wegzudenken. Hochmoderne Fertigungsanlagen[1] werden über Computer gesteuert, die ihrerseits allesamt über Microchips verfügen. Treibende Kraft in der Wirtschaft ist der Wunsch nach Steigerung der Geschwindigkeit und der Effektivität. Begleitet wird dieser Prozess durch das Ansammeln und Auswerten von Daten und Informationen in großen Mengen. Um diese Daten nutzen zu können, wurden verschiedene Algorithmen entwickelt, die komplexe Berechnungen ermöglichen. Ein Beispiel für eine solche Berechnung ist die Savings-Heuristik. Da Geschwindigkeit in der Praxis immer wichtiger für die Wettbewerbsfähigkeit von Unternehmen wird, stoßen Berechnungen in diesem Bereich teilweise an ihre Grenzen, vor allem dann, wenn auch die Qualität gewährleistet werden soll. In der Theorie könnten jederzeit optimale Rundreisen von LKW-Transporten in hoher Qualität berechnet werden. In der Praxis muss dies jedoch meist in kürzester Zeit geschehen. Soll eine schnelle Berechnung in hoher Qualität erfolgen, müssen leistungsfähigere Computer zum Einsatz kommen. Doch auch die Prozessoren von Computern stoßen in Punkto Geschwindigkeit an ihre Grenzen. Ein Prozessor ist auf sequenzielle Abläufe optimiert, was führt dazu, dass Berechnungen für eine grafische Oberfläche mit steigender Auflösung immer mehr Zeit beanspruchen, da jeder einzelne Pixel berechnet werden muss. Bei der „Full HD“ Auflösung sind es rund zwei Millionen Pixel, welche 60- bis 120-mal in der Sekunde berechnet werden müssen. Da diese Berechnung sehr viel Leistung von Seiten des Prozessors in Anspruch nimmt, wurden zur Entlastung externe Grafikkarten entwickelt. Auf diese Hardware-Erweiterung des Computers wurde ein eigener Grafikprozessor gebaut. Umgangssprachlich hat sich die englische Abkürzung von „graphics processing unit“ durchgesetzt, die GPU. Detaillierter wird auf die GPU im Kapitel 2 eingegangen.

In Disziplinen außerhalb der Logistik (z.B. Kryptografie) werden schon seit längerem Grafikkarten neben den Prozessoren für Berechnungen hinzugezogen, um die Geschwindigkeit zu steigern. In der Logistik sind Berechnungen über Grafikkarten noch nicht verbreitet, zumindest ist in den öffentlich zugänglichen Publikationen hierüber wenig bis nichts zu finden. An diesem Punkt knüpft der Gegenstand der Untersuchung dieser Bachelorarbeit an.

[1] Thomas Friedli, Günther Schuh, Wettbewerbsfähigkeit der Produktion an Hochlohnstandorten

1.2 Zielsetzung und Vorgehensweise

Ziel der Bachelorarbeit ist es, herauszufinden, ob das Verfahren der parallelen Berechnungen über Grafikkarten auch für den Bereich der Logistik sinnvoll sein kann. Zu diesem Zweck wird exemplarisch das in der Logistikwelt sehr „prominente“ Tourenplanungsproblem als Berechnungsgegenstand genommen. Hierbei wird mit der Savings-Heuristik eine zulässige Lösung erstellt.

Um eine Aussagefähigkeit bezüglich der Grafikkartenmethode zu erhalten, soll diese zwei anderen Methoden gegenübergestellt werden, welche derzeit im Bereich der Informatik als Standard gelten. Insgesamt ergeben sich somit drei verschiedene Verfahrensweisen, die näher untersucht bzw. verglichen werden:

1. Serielle Berechnung auf einem Prozessor
2. Parallele Berechnung auf einem Prozessor mit mehreren Kernen
3. Parallele Berechnung auf einer Grafikkarte
4.

In allen drei Fällen gilt es, das gleiche vorher fest definierte Ergebnis zu erreichen bzw. bei allen drei Wegen gilt es, die gleiche Qualität zu erreichen. Wird über den dritten Weg (Grafikkarte) dasselbe Ergebnis in kürzerer Zeit erreicht, als bei den anderen beiden Verfahren, lohnt es sich über Einsatzmöglichkeiten dieser Methodik in der Logistik nachzudenken.

Um die verschiedenen Verfahren miteinander vergleichen zu können, bedarf es einer bestimmten Software. In der vorliegenden Arbeit wird zu diesem Zweck ein Programm in der Programmiersprache C entwickelt, welches die serielle Berechnung, der Savings Heuristik von Clarke und Wright, mit der parallelisierten Version vergleicht. Dabei soll untersucht werden, ob eine parallele Programmierung mit Hilfe von NVIDIA´s CUDA®, Berechnungen über die Grafikkarte deutlich beschleunigen und in der Logistik eine Zeitersparnis erbringen kann.

Als Grundlage hierfür dienen Daten im XML Format von Openstreetmap. In unterschiedlich großen Netzwerken soll eine zulässige Rundreise ermittelt und im Anschluss über das Programm QGIS visualisiert werden.

1.3 Aufbau der Arbeit

Das erste Kapitel stellt die Einleitung der Bachelorarbeit dar. Es werden die Ausgangssituation, Zielsetzung und Vorgehensweise sowie der Aufbau der Arbeit nähergebracht.

Kapitel 2 vermittelt die Grundlagen der Elektrotechnik inklusive des Aufbaus und der Funktion eines Prozessors sowie die Unterschiede von CPU (Central Processing Unit bzw. Prozessor) und GPU (Graphics Processing Unit bzw. Grafikkarte). Es werden unterschiedliche Schaltungen erläutert, die notwendig sind, damit ein moderner Computer Berechnungen durchführen kann.

Gefolgt von Kapitel 3, welches die Entwicklungsumgebung CUDA® von NVDIDIA vorstellt und auf deren Verwendung näher eingeht.

In Kapitel 4 wird auf die verwendete Savings-Heuristik eingegangen und diese im Detail erläutert sowie auf mögliche Parallelisierungen und Sortierungen eingegangen.

Der praktische Teil befindet sich in Kapitel 5. Dort wird der Aufbau des entwickelten Programmes sowie die Besonderheiten bei der Implementierung erläutert und die Visualisierung mit QGIS dargestellt.

Im letzten Kapitel befindet sich die Zusammenfassung der gesamten Bachelorarbeit.

2. Allgemeine Grundlagen der Elektrotechnik

Teil der Untersuchung sind drei unterschiedliche Berechnungswege, die ihrerseits auf unterschiedlicher Hardware basieren. Als Grundlage für den Vergleich ist die Funktionsweise der Hardwarekomponenten hilfreich. In den folgenden Kapiteln wird daher auf einige Grundlagen der Elektrotechnik eingegangen, um aufbauend hierauf Einblicke in den Aufbau eines Prozessors samt seinen Transistoren und Schaltungen zu erhalten. Um auch Elektrotechnik-Laien für die Herausforderungen des technischen Fortschritts in diesem Kontext zu sensibilisieren, soll auch ein wenig auf den historischen Kontext eingegangen werden.

2.1 Historischer Exkurs – Von Relais zu Transistoren

Im Grunde genommen beginnt die Geschichte von Computern bereits im 19. Jahrhundert. Denn ohne die Erfindung einiger namhafter Persönlichkeiten, wäre es nicht möglich gewesen, den heutigen Stand der Technik zu erreichen. Einer dieser wichtigen Personen war Joseph Henry (*17.12.1797, †13.05.1878), der als Erfinder des Relais gilt. Ein Relais schaltet mittels Magnetfeld einen Schalter, durch den ein anderer Stromkreis gesteuert werden kann.

Anfang des 20. Jahrhunderts konnte ein sehr gutes Relais bereits mehrere Schaltungen in einer Sekunde tätigen, allerdings war die Grenze durch die Trägheit der Teile schnell erreicht. Im Jahre 1905[2] entwickelte John Ambrose Fleming die sogenannte Elektronenröhre. Da die Elektronen eine deutlich geringere Masse haben, konnte man mit Hilfe der Elektronenröhre wesentlich höhere Schaltfrequenzen erreichen (mehrere Tausend Schaltungen pro Sekunde). Die höhere Frequenz ging jedoch mit einer kürzeren Lebensdauer einher.

Mitte des 20. Jahrhunderts wurden Halbleiterbauelemente, wie Transistoren und Dioden, entwickelt. Sie hatten gegenüber Relais und Elektronenröhren den Vorteil, dass durch sie wesentlich kleinere und schnellere Schaltungen aufgebaut werden konnten.

Heutzutage sind in allen elektronischen Geräten meist mehrere Tausend oder gar Millionen Transistoren verbaut. Man ist nun in der Lage, hoch komplexe Steuerungen auf immer kleiner werdenden Raum zu entwickeln. Dies macht sich die Computerindustrie zu nutzen und entwickelt immer schneller werdende Computer. Durch das Voranschreiten der Computertechnologie wurde der Bedarf an grafischen Benutzeroberflächen immer größer. Während in den 80er Jahren eine Ausgabe auf einem monochromen Bildschirm mit einer 1 Bit Farbtiefe erfolgte, können moderne Systeme Grafiken mehrere Millionen Farben kraft 24 Bit Farbtiefe darstellen. Dies führt dazu, dass ein enormer Rechenaufwand allein für die grafische Darstellung erforderlich wird.

[2] Otto Zinke und Heinrich Brunswig, Lehrbuch der Hochfrequenztechnik

Die ersten Grafikkarten waren dafür konzipiert, die bereits berechneten Grafiken auf dem Monitor auszugeben und konnten selbst keine Berechnungen durchführen. Seit Anfang des 21. Jahrhunderts können Grafikkarten mit ihrer GPU eigenständige Programme und Berechnungen ausführen.

Im folgenden Kapitel wird näher auf die Transistorschaltungen eingegangen.

2.2 Vom Transistor zum Prozessor

In Kapitel 2.1 wurde auf Bauelemente (Relais und Transistor) eingegangen, auf deren Basis nun zwei Zustände herstellbar sind: Storm fließt und Strom fließt nicht. In der Computertechnik wird dies oftmals als „true“ oder „false“ einem Bit zugeordnet. Mittels dieses Binärsystems und der booleschen Algebra können Berechnungen durchgeführt werden.

Die Abbildung 2.1 zeigt auszugsweise die wichtigsten genormten Symbole für Schaltungen in der Elektrotechnik, die auf Transistoren basieren.[3]

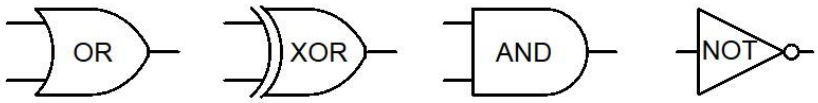


Abbildung 2.1 Symbole von links nach rechts: OR-Gate, XOR-Gate, AND-Gate, NOT-Gate

In der in Abbildung 2.2 gezeigten Schaltung können zwei Bits nach der booleschen Algebra addiert werden. Dabei wird das Ergebnis über den Ausgang als Summe ausgegeben und, falls vorhanden, ein weiteres Übertragungsbit als Rest. Diese Schaltung wird Halb-Addierer genannt.

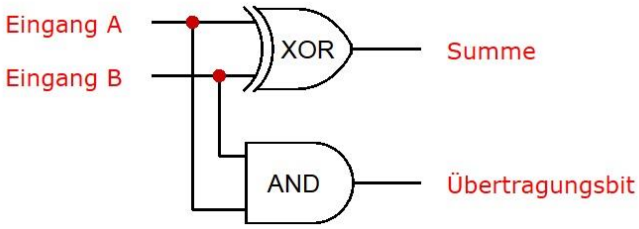


Abbildung 2.2 Darstellung eines Halb-Addierers

[3] Auf Details und ihre Funktionsweise wird an dieser Stelle nicht näher eingegangen, da dies als Grundwissen vorausgesetzt wird. Zur Vertiefung dieser kann die Norm EN 60617-2 herangezogen werden

Für moderne Computer reicht es nicht aus, in einem 1-Bit System zu rechnen. Sie laufen heutzutage über ein 32-Bit oder gar 64-Bit System. Werden zwei Halb-Addierer und ein OR-Gate, wie in Abbildung 2.3 zusammengeschaltet, erhält man einen Voll-Addierer, mit dem die Bitanzahl endlos skaliert werden kann.

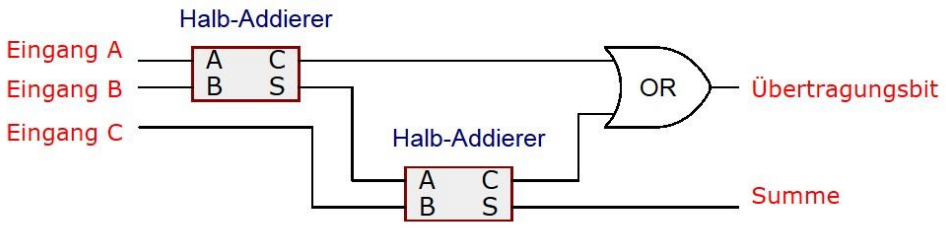


Abbildung 2.3 Darstellung eines Voll-Addierers

Im Gegensatz zu einem Halb-Addierer, hat der Voll-Addierer einen Eingang für das Übertragungsbit, mit dessen Hilfe er die nächste Stelle der binären Zahl berechnen kann. Abbildung 2.4 zeigt beispielhaft einen 4-Bit-Addierer, unter dessen Verwendung zwei vierstellige Bitzahlen addiert werden können.

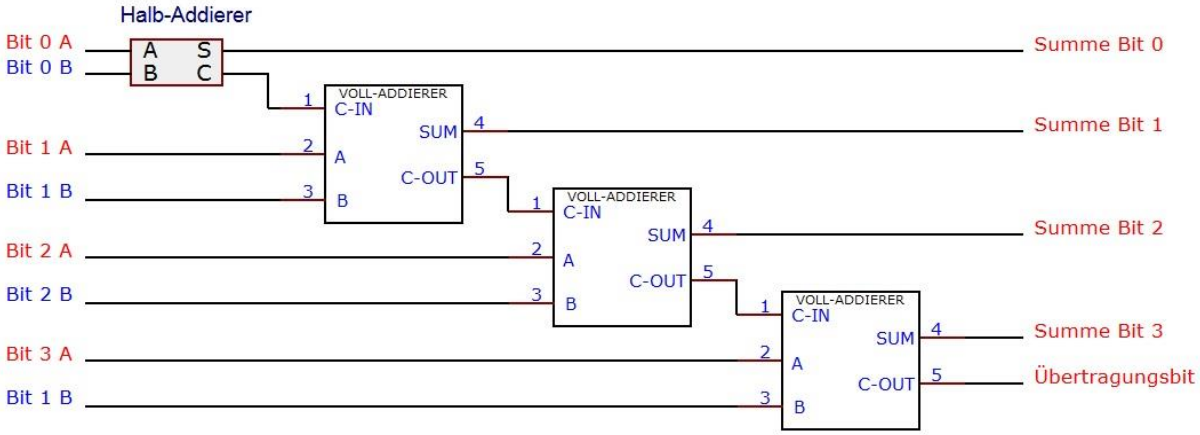


Abbildung 2.4 Schaltung eines 4-Bit-Addierer

Solche Addierer sind in allen Computerprozessoren verbaut. Sie sind ein Teil der arithmetisch-logischen Einheit, im Englischen „Aritmetic Logic Unit“ genannt bzw. abgekürzt auch ALU. Die Halb- und Voll-Addierer bilden den arithmetischen Teil. Der logische Part wird durch weitere Schaltungen, wie zum Beispiel Subtrahieren oder Summe, gestellt. Welche Operation die ALU durchführen soll, wird über einen Befehlscode der Kontrolleinheit gesteuert. Informatiker sprechen vom Maschinencode.

Die ALU, die Kontrolleinheit und die verschiedenen Speicherregister bilden zusammen den Hauptprozessor eines Computers, im englischen „Central Processing Unit“, kurz CPU genannt. Auf diesen wird im nächsten Kapitel näher eingegangen.

2.3 Central Processing Unit (CPU)

Im Kapitel 2.2 wurde bereits dargestellt, dass eine CPU aus mehreren Bauteilen besteht. Diese unterscheiden sich zwar je nach Hersteller und Baureihe, doch greifen fast alle Prozessoren auf den „von Neumann Zyklus“ zurück. Er besteht im Wesentlichen aus drei Teilschritten, die hier betrachtet werden sollen.

- Befehlsabruf „FETCH“, aus dem Speicher wird der nächste Befehl geladen
- Dekodierung „DECODE“, der Befehl wird in eine Schaltinstruktion aufgelöst
- Befehlsausführung „EXECUTE“, der Befehl wird ausgeführt

Anhand der Taktung eines Prozessors wird vorgegeben, in welcher Geschwindigkeit diese Teilschritte nacheinander ausgeführt werden. In Abbildung 2.5 wird dies schematisch dargestellt.

Abbildung 2.5 schematische Darstellung von Neumann Zyklus

Bei dieser Bauart kann in jedem dritten Takt ein Befehl ausgeführt werden. Da jeder Teilschritt von einem anderen Bauteil ausgeführt wird, führt dies in der Theorie dazu, dass sich in zwei von drei Takten ein Bauteil im Leerlauf befindet. Ingenieure haben dieses Problem durch teilweise Parallelisierung der Teilschritte gelöst (siehe Abb. 2.6).

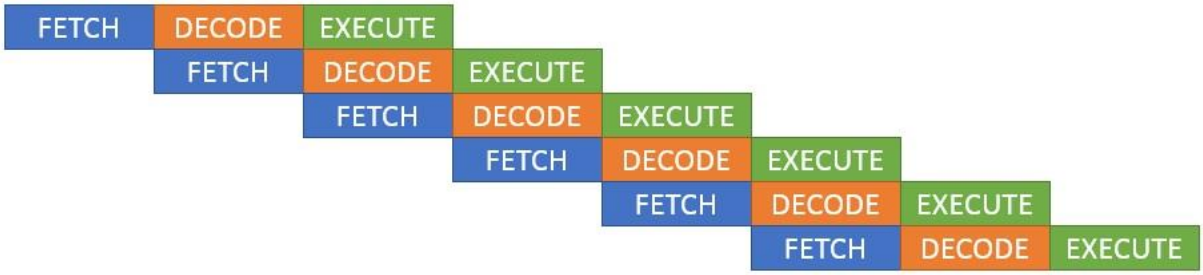


Abbildung 2.6 modifizierte schematische Darstellung von Neumann Zyklus

Durch Parallelisierung ist es möglich, jeden Takt der CPU einen Befehl auszuführen zu lassen. Gute Prozessoren laufen heutzutage bei rund 4 Gigahertz, kurz GHz, dem entsprechen $4*10^9$ Befehle pro Sekunde. Eine höhere Taktung ist generell möglich, würde aber zu einer höheren Wärmeentwicklung führen, die wiederum einen erhöhten Aufwand bei der Kühlung verursacht. Damit dennoch mehr Leistung aus dem Computer herausgeholt werden kann, wurden sogenannte Mehrkernprozessoren entwickelt. Hierbei handelt es sich um mehrere Kerne, die jeweils eine CPU haben, die parallel in einem Prozessor arbeiten. In Abbildung 2.7 ist schemenhaft ein Prozessor inklusive seiner vier Kernen zusehen.

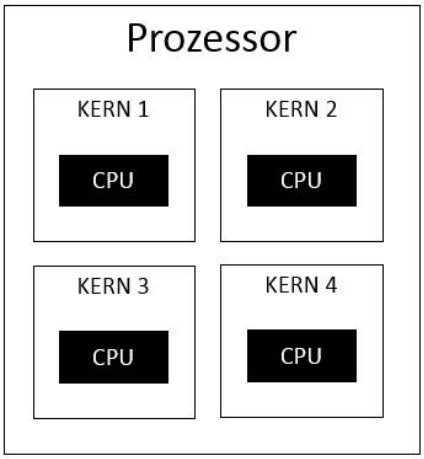


Abbildung 2.7 schematische Darstellung Quadcore Prozessor

Aufgrund dieser Bauart konnten bereits vier Befehle zur selben Zeit ausgeführt und damit die Leistung der Computer weiter gesteigert werden.

Aus der Sicht der Softwareentwickung ergab sich beim damaligen Markteintritt dieser Prozessoren insofern ein Nachteil, als dass die Betriebssysteme oder andere Programme, die es zu der Zeit gab, nicht für symmetrische Multiprozessorsysteme, kurz SMP, ausgelegt waren. Damit der Vorteil der Quadcore Prozessoren ausgeschöpft werden konnte, mussten die Softwareentwickler erst nacharbeiten bzw. dafür sorgen, dass alle Kerne der Prozessoren ausgenutzt wurden. Heute sind Quadcore Prozessoren im Prinzip schon Standard.

Der Computerchiphersteller Intel entwickelte zusätzlich zu den Mehrkernprozessoren, eine sogenannte Hyper-Threading-Technik, kurz HTT. In dieser hat jede CPU nicht nur eine ALU, sondern gleich zwei oder mehr, wodurch die Leistung erneut gesteigert werden konnte. Allerdings gilt auch hier: für die volle Leistung, musste die Software das SMP unterstützen. In Abbildung 2.8 ist der von Neumann Zyklus eines Kernes mit zwei ALUs im HTT zu sehen.

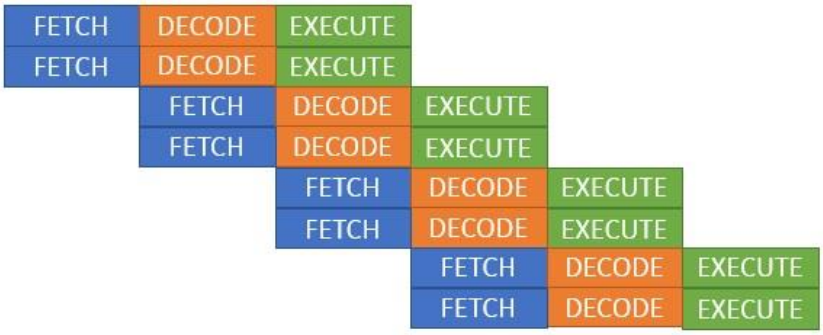


Abbildung 2.8 Intel Hyper-Threading-Technik

Schaut man sich die genaue Spezifikation eines Intel Core i7 6700K an, werden dort folgende Werte angegeben:

- Ein Prozessor
- Vier Kerne
- Maximal acht Threads die parallel laufen können
- 4.00 GHz

Da der Prozessor über die Hyper-Threading-Technik verfügt, ist er in der Lage

$$1 * 4 * 8 * 4 * 10^9 = 128 * 10^9$$

Befehle pro Sekunde auszuführen.

Alle Prozessoren sind so konstruiert, dass sie eine Verzögerung minimieren und ihre Befehle schnellstmöglich ausführen, dabei aber hochkomplexe Aufgaben bewerkstelligen können. Damit diese Menge an Daten bei den hohen Geschwindigkeiten zur Verfügung steht, sind in einer CPU eigene Speicher integriert, denn die physische Entfernung zum „normalen“ Arbeitsspeicher genügt bereits, um die Datenübertragung zur CPU zu verzögern.

Der interne Speicher einer CPU wird Cache genannt, in diesem gibt es ein mehrstufiges System. Je nach Prozessorarchitektur kann dieses eine bis vier Stufen, genannt Level (kurz L), aufweisen. Dabei ist der L1 Cache der ALU am dichtesten und auch am schnellsten. Je höher die Zahl des L Cache wird, desto weiter weg und langsamer wird er. In Abbildung 2.9 ist der Benchmark der Speicher eines Intel Core i7 6700k zu sehen. Das Benchmark wurde mittels der frei erhältlichen Software AIDA64[4] erstellt. Es ist deutlich ablesbar, dass der Arbeitsspeicher, hier als „Memory" gekennzeichnet, wesentlich langsamer ist, als die L1, L2 und L3 Speicher. Desweiteren sind auch die Unterschiede und Abstufungen des Cache gut erkennbar. Da der Arbeitsspeicher außerhalb des Prozessors liegt und auch langsamere Zugriffszeiten hat, wird dieser Speicherzugriff zum Flaschenhals aller Anwendungen und Berechnungen.

	Read	Write	Copy	Latency
Memory	31251 MB/s	32585 MB/s	29950 MB/s	66.1 ns
L1 Cache	1043.7 GB/s	525.19 GB/s	1047.6 GB/s	1.0 ns
L2 Cache	462.61 GB/s	287.09 GB/s	394.59 GB/s	3.0 ns
L3 Cache	276.29 GB/s	179.66 GB/s	230.64 GB/s	14.3 ns

Abbildung 2.9 Auszug AIDA64 Speichertest vom Intel i7

Das für Speicherstandards zuständige Joint Electronic Devices Engineering Council hat für das Jahr 2019 bereits angekündigt, einen neuen Standard einzuführen. Mit diesem DDR5-Standard soll die Geschwindigkeit des Arbeitsspeichers verdoppelt werden und bis zu 51 Gigabyte je Sekunde erlauben.

2.4 Graphics Processing Unit (GPU)

Der größte und ausschlaggebendste Unterschied zwischen einer CPU und einer GPU ist die Anzahl der ALUs. Während die CPU nur ein bis zwei ALUs hat, kommen in der GPU mehrere hundert zum Einsatz, durch die die Berechnungen durchgeführt werden können. Der Grafikkartenhersteller NVIDIA nennt diese intern „CUDA-Core", andere Hersteller sprechen von Streaming-Prozessoren. In Abbildung 2.10 ist eine schemenhafte Darstellung vom unterschiedlichen Aufbau einer CPU und einer GPU inklusiver ihrer ALUs zu sehen. Die aktuellsten Grafikkarten im Jahre 2017 besitzen schon mehrere Tausend Streaming-

[4] https://www.aida64.com/

Prozessoren. Zu erwähnen sind auszugweise die Grafikkarten vom Typ „AMD Radeon RX Vega“ mit 4096 Streaming-Prozessoren und die „NVIDIA Geforce GTX Titan Z“, welche über 5760 CUDA-Cores verteilt auf mehreren Streaming-Multiprozessoren verfügt.

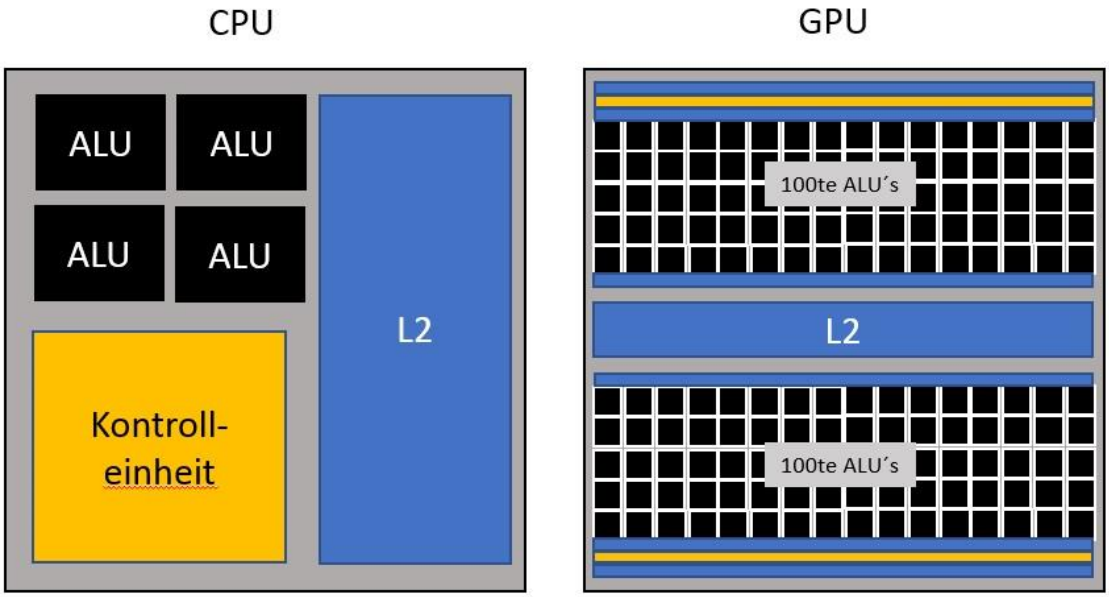


Abbildung 2.10 Vergleich der ALU Aufbauschemen

NVIDIA gibt folgende Spezifikation der Geforce Titan Z an:

- 5760 CUDA-Cores
- Basistakt 705 MHz

Daraus ergibt sich eine theoretische Anzahl an Berechnungen von

$$5760 * 705 * 10^6 = 4.060.800 * 10^6$$

Mit rund $4 * 10^{12}$ Rechenoperationen pro Sekunde kann dieser moderne Grafikprozessor rund 30-mal mehr in derselben Zeit berechnen, wie der in Kapitel 2.3 erwähnte Intel i7.[5]

[5] Diese extremen Rechenleistungen lassen sich auch für andere Berechnungen außer Grafiken nutzen. Ein sehr verbreiteter Anwendungsbereich ist beispielsweise die Kryptografie. Hier sorgte jüngst die digitale Währung Bitcoin – auch Kryptowährung genannt – für große Aufmerksamkeit in den Medien. Bitcoins werden durch einen Hash-Algorithmus generiert. Eine der größten Bitcoinfarmen der Welt steht in Island (Genesis Mining Farm). In dieser arbeiten mehrere 10.000 Grafikkarten parallel zusammen als eine Recheneinheit, kraft derer die Währung generiert wird.

Diese Leistung kann jedoch nur bei *parallelen* Berechnungen erreicht werden und auch nur dann, wenn die zu berechnenden Daten unabhängig voneinander sind. Bei *seriellen* Berechnungen mittels einer GPU ist die Geschwindigkeit geringer als bei einer CPU, denn es können theoretisch nur noch 705 * 10^6 Berechnungen pro Sekunde durchgeführt werden.

Zur weiteren Leistungssteigerung können auch mehrere Grafikkarten zusammengeschaltet werden, dies wir bei NVIDIA über die „Scalable Link Interface“, kurz SLI Technik verwirklicht.[6] Das Zugreifen auf die Grafikkarte im Rahmen der Softwareentwicklung wird heutzutage durch vorgefertigte Bibliotheken, wie zum Beispiel „OpenCL“, sichergestellt. NVIDIA hat für diesen Zweck 2006 für seine Grafikkarten eine eigene Programmierschnittstelle hergestellt und stellt diese kostenfrei zur Verfügung: „Compute Unified Device Architecture“ kurz CUDA®, auf das im nächsten Kapitel näher eingegangen wird.

[6] Auf den Aufbau und die Funktion der SLI wird hier nicht näher eingegangen. Näheres hierzu gibt es unter http://www.nvidia.de/object/sli-technology-overview-de.html

3. Zur Funktionsweise von Compute Unified Device Architecture (CUDA®)

Während im vorherigen Kapitel detailliert auf den Aufbau und die Funktion der CPU sowie der GPU eingegangen wurde, befasst sich dieses Kapitel mit der Programmierschnittstelle CUDA®.

Bei CUDA® handelt es sich um eine Sammlung an Bibliotheken und eine Programmierschnittstelle, die es der Softwareentwicklung ermöglicht, die GPU für Zwecke außerhalb der visuellen Berechnungen als Ko-Prozessor zu nutzen. CUDA® kommt im Praxisteil der Arbeit zum Einsatz, um zu untersuchen, ob durch parallele Berechnungen mit Hilfe der Grafikkarte, ein logistisches Problem schneller gelöst werden kann. CUDA® wurde aufgrund der zur Verfügung stehenden Hardware, der guten Dokumentation und der leichten Implementierung für diese Arbeit gewählt. Alternativ kann dieses auch durch eine andere Programmierschnittstelle, wie zum Beispiel OpenCL, erreicht werden.

3.1 Kurz über CUDA

CUDA® wird mit einer Softwareumgebung geliefert, die es Entwicklern ermöglicht, C als Highlevel Programmiersprache zu verwenden. Auch andere Sprachen wie Fortran und Java werden unterstützt, auf die in dieser Arbeit jedoch nicht näher eingegangen wird, da der Umfang den Rahmen dieser Bachelorarbeit überschreitet.

Infolge des Aufkommens von Multiprozessorsystemen und GPUs und deren vielen Streaming-Prozessoren wurden die modernen Computer zu parallelen Systemen. Die Herausforderung für Softwareentwickler besteht hierbei darin, die Parallelität in der Anwendung durch die steigende Prozessorzahl beizubehalten und skalieren zu lassen. Hierfür wird häufig auf das Gustafsons-Gesetz verwiesen, welches im Kern die Zeit für die Lösung eines Problems festlegt und mit steigender zur Verfügung stehender Prozessorzahl die Größe des Problems ansteigen lässt. Sei tP der Anteil der Laufzeit des parallelen Abschnittes eines Programmes, dann er gibt sich aus $(1 - tP)$ der sequenzielle Anteil und eine gesamte Laufzeit auf einem Prozessor von

$$1 = (1 - tP) + tP$$

Werden Speicherzugriffzeiten und andere Verzögerungen durch Kommunikation oder Synchronisation vernachlässigt, kann der parallele Anteil auf N Prozessoren gleichzeitig laufen und dadurch den Beschleunigungsfaktor $B(N)$ erhöhen. Somit gilt

$$B(N) = (1 - tP) + N * tP$$

Als Beispiel sei hier das Rendern von 3D-Grafiken zu erwähnen. Möchte man fest 30 Bilder je Sekunde ausgeben, können mittels steigender Prozessorzahlen größere Datenmengen verarbeitet werden und damit eine wesentlich detailliertere Grafik, durch zum Beispiel mehr Polygone, erreicht werden. Wie in Abbildung 3.1 zu sehen, ist es mit CUDA® möglich, eine Applikation so aufzuteilen, dass die CPU und die GPU parallel unterschiedliche Aufgaben abarbeiten. CUDA® erweitert die Programmiersprache C um drei Schlüsselbereiche:

- Hierarchie von Thread-Gruppen
- Speicherverwaltung der GPU
- Barrieren für die Synchronisation

Auf diese drei Schlüsselbereiche wird im nächsten Kapitel näher eingegangen.

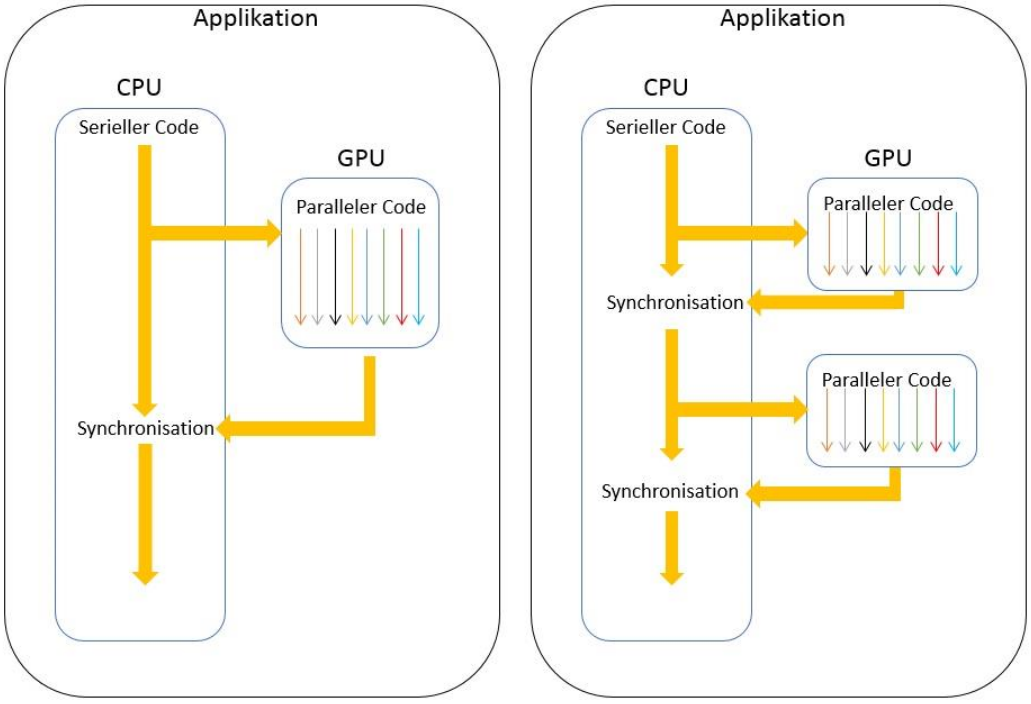


Abbildung 3.1 beispielhafte Aufbauten von CUDA® Applikationen

3.2 Thread-Gruppen Hierarchie

Während bei Programmiersprachen, wie zum Beispiel C/C++ oder Java, von Funktionen gesprochen wird, werden diese in CUDA® als „Kernel“ bezeichnet. Wie aus Abbildung 3.2 hervorgeht, werden die Kernels, genau wie Funktionen, vor dem Hauptprogramm definiert. Dabei wird dem Kernel beim Deklarieren ein Spezifizierer vorangestellt. Dieser legt fest, ob der Kernel auf der CPU oder der GPU ausgeführt wird und ob er von der CPU oder GPU aufgerufen werden kann. Auszugsweise wird hier *__global__* und *__device__* erläutert.

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
int main()
{
...
// Kernel Aufruf mit N Threads
VecAdd<<<1, N>>>(A, B, C);
...
}
```

Abbildung 3.2 Kernel zum Summieren von 2 Vektoren

Mittels *__device__* ist der Kernel nur von der GPU auf der GPU ausführbar. *__global__* wird von der CPU aufgerufen, aber von der GPU ausgeführt. Modernere Grafikkarten, die eine CUDA® Version 3.2 oder höher aufweisen, können diese Kernels auch von der GPU aus aufrufen. Näheres dazu gibt NVIDIA unter *„CUDA Dynamic Parallelism“* an. Weitere Spezifizierer sind der CUDA® Dokumentation zu entnehmen.[7]

Die kleinstmögliche Instanz ist ein „Thread“. Ein Thread führt ein Kernel direkt auf einem CUDA-Core aus. Mehrere Threads werden in einem „Block“ zusammengefasst. Ein Block wird auf einem Multiprozessor ausgeführt, ein Multiprozessor besitzt mehrere CUDA-Cores. Dabei werden die Blöcke von Threads immer auf alle Multiprozessoren gleichmäßig verteilt. In Abbildung 3.3 sind zwei mögliche Aufteilungen der Blöcke dargestellt. In diesem Beispiel werden acht Blöcke auf dem vorhandenen Streaming-Multiprozessor (SM) aufgeteilt. Es ist zu erkennen, dass eine erhöhte Anzahl an Prozessoren die Ausführung beschleunigt.

[7] http://docs.nvidia.com/cuda/

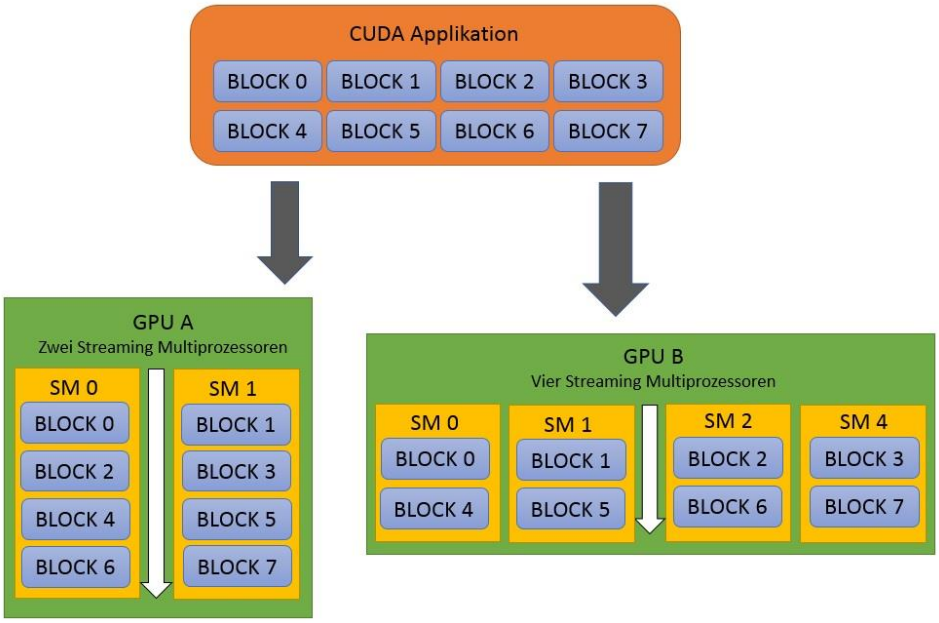


Abbildung 3.3 Blöcke werden auf die Multiprozessoren verteilt

Mehrere Blöcke werden zusammengefasst zu einem „Grid“. Somit wird ein Kernel von einem Grid mit n Blöcken zu je m Threads ausgeführt. CUDA® stellt zur genauen Identifizierung eines einzelnen Threads einen Drei-Komponenten-Vektor zur Verfügung, über den ein globaler Thread-Index i, in eindimensional, zweidimensional oder dreidimensional erstellt werden kann. Hierzu werden, wie in Abbildung 3.4, die variablen *blockIdx* für den Index eines Blocks in einem Grid, die *blockDim* für die Anzahl der Threads in einem Block sowie die *threadIdx* für den Index eines Threads innerhalb eines Blocks herangezogen. Somit gilt

$$i = blockIdx.x * blockDim.x + threadIdx.x$$

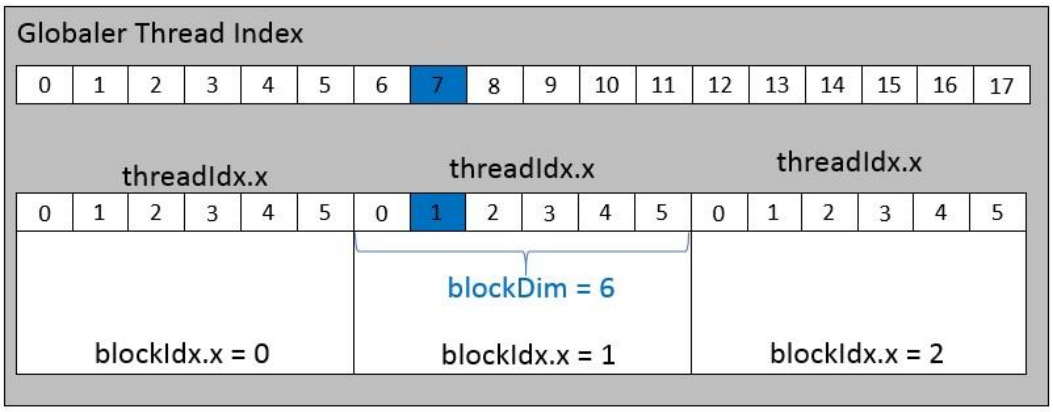


*Abbildung 3.4 globaler Index i = 7 ergibt sich aus i = 1 * 6 + 1*

Die maximale Anzahl[8] ist auf 1024 Threads pro Block begrenzt, dabei kann ein Kernel von mehreren Blöcken ausgeführt werden.

Die Anzahl an Blöcken innerhalb eines Grids ist bei eindimensionalen auf 2.147.483.647, bei zweidimensionalen auf 2.147.483.647 * 65.535 und bei dreidimensionalen auf 2.147.483.647 * 65.535 * 65.535 Blöcken begrenzt. Wird diese mit der maximalen Anzahl von 1024 Threads pro Block multipliziert, ergibt sich bei eindimensionalen Grids eine Instanzgröße von rund 2199 * 10^9 Threads, welche von einem Kernelaufruf ausgeführt werden können. Bei dreidimensionalen Grids sind es sogar 9223 * 10^{15} Threads. Allerdings stößt man bei diesen Dimensionen an die Grenze eines 32 Bit integer Wertes, der bei 2.147.483.647 liegt.

Sollten dennoch Instanzen genutzt werden, die größer sind, muss dies beim Reservieren des Speichers beachtet und dem Datentypen des globalen Indexes angepasst werden.

Erhält ein Multiprozessor einen oder mehrere Blöcke zum Ausführen, werden diese in jeweils 32 Threads zu einem sogenannten Warp zerlegt. Ein Warp führt einen Befehl immer gleichzeitig aus. Eine maximale Effizienz wird erreicht, wenn die Anzahl der Threads pro Block durch 32 dividierbar ist, dadurch wird ein Warp komplett gefüllt. Die Anzahl der Warps Wn lässt sich wie folgt berechnen: sei n Anzahl der Threads, und Warpgröße $Wx = 32$.

$$Wn = \left\lceil \frac{n}{Wx} \right\rceil$$

Threads können während ihrer Ausführung auf Daten verschiedenster Speicherbereiche zugreifen, worauf im folgenden Kapitel näher eingegangen wird.

3.3 Speicherhierarchie und Synchronisation

Nach dem im vorherigen Kapitel detailliert auf Grids, Blöcke, Threads und Warps eingegangen wurde, befasst sich dieses Kapitel mit der Speicherhierarchie und Verwaltung. Während eine CPU, im folgenden Host genannt, ihren eigenen Cache sowie den Arbeitsspeicher zur Verfügung hat, besitzt eine GPU auf der Grafikkarte, im folgenden Device genannt, ihre eigene Speicherstruktur. Wie in Abbildung 3.5 dargestellt, hat jeder einzelne Thread einen lokalen Speicher und jeder Block einen geteilten Speicher für die im Block vorhandenen Threads. Darüber hinaus steht ein globaler Speicher zur Verfügung, auf den alle Threads gemeinsam zugreifen können.

[8] Stand 2017

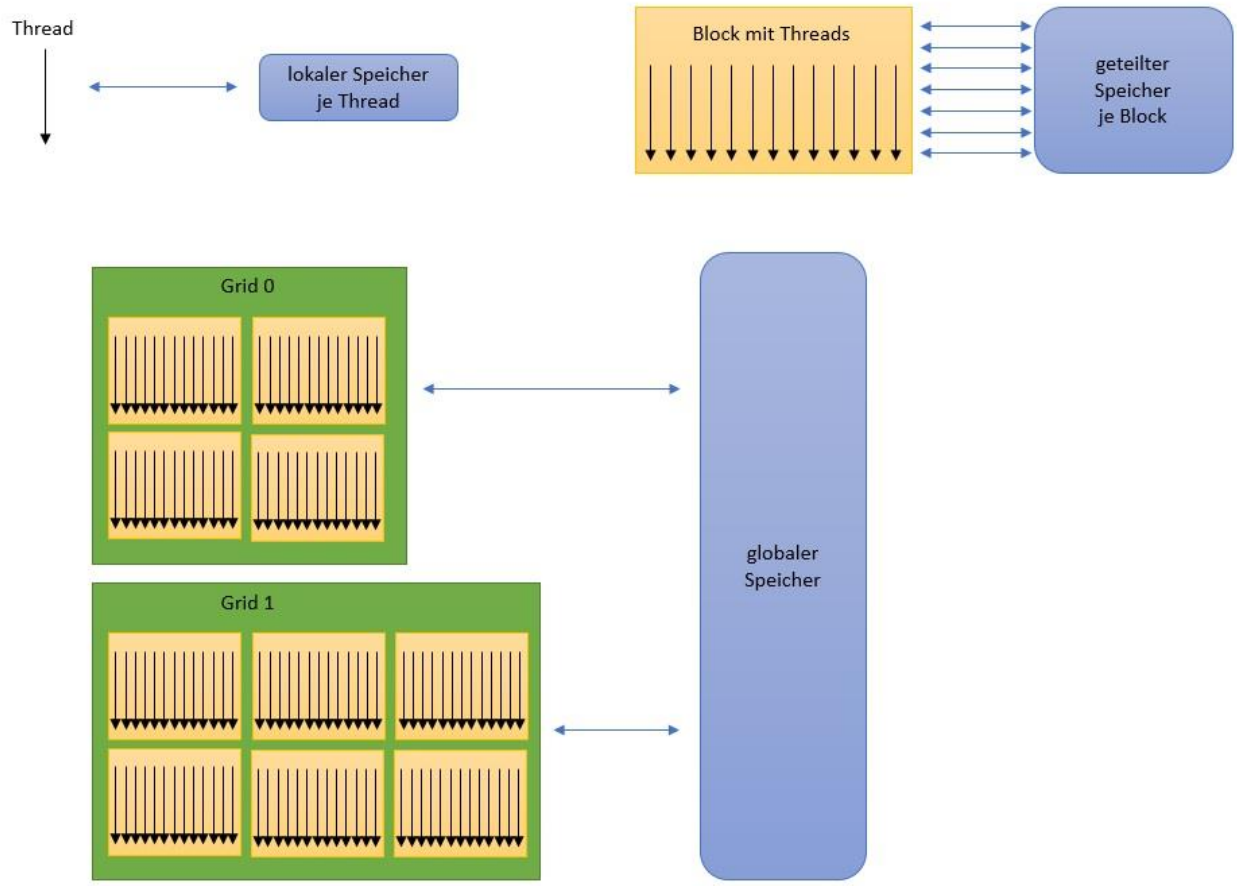


Abbildung 3.5 Speicherhierarchie von CUDA®

In welchem Speicher die Variablen abgelegt werden, legt der Entwickler im Kernel fest (vgl. Abbildung 3.6). Dabei ist zu beachten, dass der geteilte Speicher dem lokalen und globalen vorzuziehen ist, da dieser der performanteste ist. Eine detailliertere Auflistung zu den Geschwindigkeiten stellt NVIDIA in der CUDA® Dokumentation zur Verfügung und soll hier nicht weiter behandelt werden.[9]

[9] http://docs.nvidia.com/cuda/

```
//Kernel Definition
__global__ void Add(int * A, int * B, int * C)
{
int var;  //Lokale Variable
__shared__ int s_var;  //Variable im geteilten Speicher
__device__ int g_var;  //Variable im globalem Speicher

}

int main()
{
...
}
```

Abbildung 3.6 Deklaration von Variablen in CUDA®

Da die Reihenfolgen der Speicherzugriffe im geteilten und globalen Speicher nicht definiert sind, ist ohne eine explizite Synchronisation nicht festgelegt, wann welcher Thread auf denselben Speicherbereich zugreift. Dem wird durch Barrieren entgegenwirkt, die CUDA® zur Verfügung stellt. Diese werden zum Beispiel beim Schreiben von Daten in den geteilten Speicher benötigt. Per Funktion *__syncthreads()* im Kernel muss sichergestellt werden, dass alle Threads ihre Daten in den geteilten Speicher geschrieben haben, ansonsten ist nicht gewährleistet, dass auf den Speicherbereich zugegriffen wird, bevor dort Daten hinterlegt sind.

Eine weitere wichtige Synchronisation ist beim Kopieren von Daten aus dem Host-Speicher in den Device-Speicher notwendig. Hierbei ist sicherzustellen, dass alle Daten auf dem Device liegen, bevor der Kernel aufgerufen wird. Dies wird durch die Funktion *cudaDeviceSychronize()* im Hauptprogramm erreicht. Zur Leistungsoptimierung, sollten drei grundlegende Strategien angewendet werden:

- Parallele Ausführung maximieren
- Maximalen Speicherdurchsatz durch optimierte Speichernutzung
- Instruktionsdurchsatz durch Nutzung von intrinsischen Funktionen maximieren

Welche Strategie zu dem größten Leistungszuwachs führt, hängt von der Leistungsbegrenzung ab. Ein Kernel, der hauptsächlich durch Speicherzugriffe begrenzt wird, wird keine signifikante Leistungssteigerung durch die intrinsische Funktion erlangen. Daher steht bei der Optimierung das Messen und Überwachen der Leistungsbegrenzer im Fokus. Hierfür können die Angaben der Hersteller, wie zum Beispiel „Floating Point Operations Per Second“, kurz FLOPS, als Vergleich herangezogen werden, um die maximale Ausnutzung der Hardware zu erreichen. Grundlegend ist festzuhalten, dass das Kopieren von Daten vom

Host zum Device, sowie vom Device zum Host am meisten Leistungseinbuße mit sich bringt, da die Schnittstelle der Grafikkarte in der Version 3.0 für eine maximale Übertragungsrate von rund 15 Gigabyte je Sekunde ausgelegt ist. Daher sollte die Kopierfrequenz minimiert, aber die Datenmenge beim Kopieren maximiert werden.

Es wird empfohlen folgenden Programmablauf einzuhalten

1. Speicher auf dem Device reservieren, *cudaMalloc()*
2. Daten vom Host zum Device kopieren, *cudaMemcpy()*
3. Synchronisieren, *cudaDeviceSynchronize()*
4. Kernel starten
5. Synchronisieren, *cudaDeviceSynchronize()*
6. Daten vom Device zum Host kopieren, *cudaMemcpy()*
7. Synchronisieren, *cudaDeviceSynchronize()*
8. Speicher auf dem Device freigeben, *cudaFree()*

Nachdem das Programm fertig geschrieben wurde, musste der Code in Maschinensprache kompiliert werden. Hierauf wird im nächsten Kapitel kurz eingegangen.

3.4 Kompilation

Damit ein Quellcode zum Einsatz kommen kann, muss er in eine für Maschinen verständliche Sprache übersetzt werden. Dieses Verfahren wird Kompilieren genannt. Für die Programmiersprache C ist in CUDA® ein eigener Compiler namens *nvcc* integriert, dies ist eine von NVIDIA modifizierte Version der „GNU Compiler Collection“. Benutzt wird dieser ähnlich, wie der GCC oder MinGW. Eine andere Alternative zum manuellen Kompilieren über eine Konsole bietet *Microsoft Visual Studio*. Beim Installieren von CUDA® werden alle benötigten Bibliotheken und Verknüpfungen automatisch implementiert und über Vorlagen bereitgestellt. Eine ausführliche Anleitung und detaillierte Informationen zum Kompilieren bietet das Kapitel *„Programming Interface“* in der CUDA® Dokumentation[10] von NVIDIA.

Im nächsten Kapitel wird die Implementierung von CUDA®, im Rahmen einer logistischen Aufgabe aufgezeigt und die benötigte Berechnungszeit zwischen serieller und paralleler Programmierung mit CUDA® verglichen.

[10] http://docs.nvidia.com/cuda/

4. Anwendungsmöglichkeiten von parallelen Berechnungen am Beispiel einer Heuristik

In dem vorangegangenen Kapitel wurde die Grundlage für die Implementierung und die Nutzung von NVDIA CUDA® geschaffen.

Damit die Effektivität dieser Technik in der Logistik genauer untersucht werden kann, wurde für diese Arbeit ein eigenes Programm entwickelt, mit dessen Hilfe die Berechnungszeiten zwischen CPU und GPU vergleichen werden können. Hierbei wird in den folgenden Kapiteln eine Rundreise unter Verwendung der „Savings-Heuristik“ erstellt. Als Ausgangsdaten für die Entfernungen werden XML Dateien von Openstreetmap genutzt. Diese sind frei erhältlich und beinhalten unter anderem GPS[11] Koordinaten von Flughäfen. An dieser Stelle wird auf die vorangegangene Studienarbeit „Bestimmung optimaler Rundreisen mit realen Openstreetmap-Daten und Visualisierung im Geoinformationssystem QGIS“ verwiesen.

4.1 Saving-Heuristik

Die Savings-Heuristik wurde im Jahr 1964 unter dem Namen „*Scheduling vehicles from a central delivery depot to a number of delivery points*„ das erste Mal von Clarke und Wright veröffentlicht. Bei dieser Heuristik wird davon ausgegangen, dass von einem Depot aus einzelne Kunden sternförmig angefahren werden. Die einzelnen Touren werden auf eine mögliche Verbesserung, durch zusammenlegen zweier Subtouren, hin untersucht.

In dem folgenden Beispiel wird ein ungerichtetes Netzwerk herangezogen. In diesem sind die Hin- und Rückfahrten identisch lang und zur Vereinfachung werden keine weiteren Restriktionen, wie zum Beispiel Abbiegeverbote, Einbahnstraßen, Fahrzeiten oder Kapazitäten berücksichtig.

Als erstes müssen alle Knoten (im folgenden Kunden genannt) mit einem Startknoten (folgend Depot) verbunden werden. Zum Erstellen einer Subtour wird die Hin- und Rückfahrt aufsummiert und als Kantenwert $subT_n$ gespeichert, dabei entstehen $n-1$ Subtouren. Es gilt eine Kantenbewertung

$$subT_n = t_{depot,n} + t_{n,depot}.$$

Im zweiten Schritt werden sogenannte „Savings“, zu Deutsch Einsparungen, berechnet. Über das Saving wird die eingesparte Entfernung ausgedrückt, die man durch das Zusammenlegen zweier Subtouren wie in Abbildung 4.1 zu sehen ist, erhält. Sei $subT_1$ die Subtour Depot -

[11] Global Positioning System

Kunde 1 – Depot und $subT_4$ Depot - Kunde 4 – Depot, dann berechnet sich das Saving $S_{1,4}$ durch

$$S_{1,4} = subT_1 + subT_4 - (t_{depot,1} + t_{1,4} + t_{4,depot})$$

Anders ausgedrückt

$$S_{1,4} = t_{depot,1} + t_{1,depot} + t_{4,depot} + t_{depot,4} - t_{depot,1} - t_{1,4} - t_{4,depot}$$

Daraus ergibt sich

$$S_{1,4} = t_{1,depot} + t_{4,depot} - t_{1,4}$$

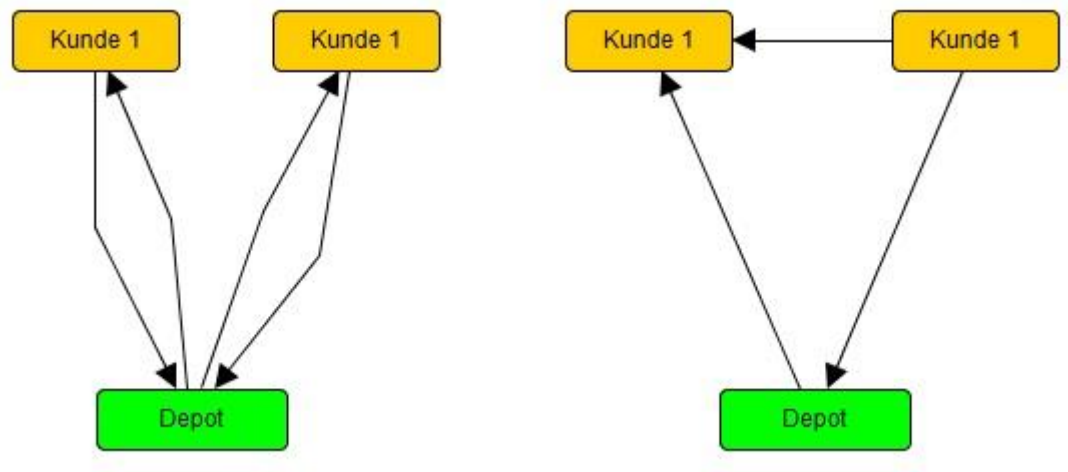


Abbildung 4.1 links einzelne Pendeltouren, rechts verbundene Subtouren

Die Anzahl der Savings kann mittels $x = \frac{n*(n-1)}{2} - n + 1$ errechnet werden. Im Anschluss werden die Savings, je nach Implementierung, in auf- oder absteigender Reihenfolge sortiert. Dabei ist zu beachten, dass die Nummer der beiden Kunden, die zu dem Saving gehören, mitgespeichert werden, um später darauf zurückgreifen zu können. Eine Möglichkeit hierzu wird in dem Kapitel 5.3 durch die globale Id vorgestellt. Nach dem alle Pendeltouren und Savings berechnet sind, wir nun die Rundreise, wie in Abbildung 4.2 dargestellt, nachfolgenden Regeln erstellt:

1. Verbinde zwei Subtouren miteinander, die das größte Saving ermöglichen, wenn
    a. beide Kunden jeweils noch mindestens eine Kante zum Depot haben
    b. beide Kunden sich in unterschiedlichen Subtouren befinden
2. Wiederhole Schritt 1, solange noch positive Savings vorhanden sind

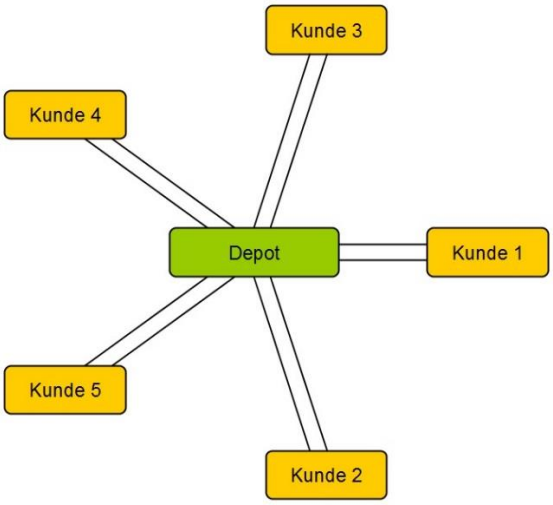
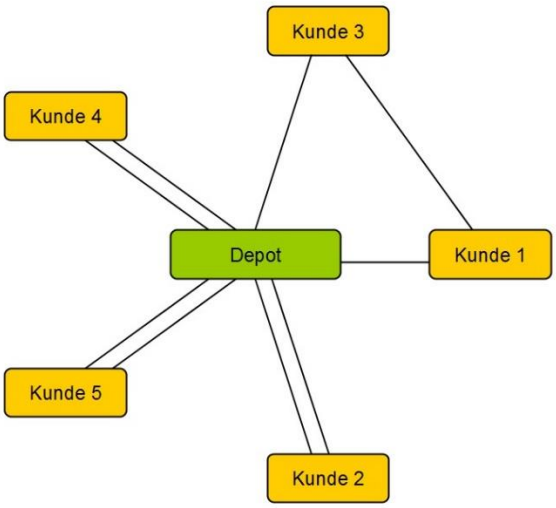
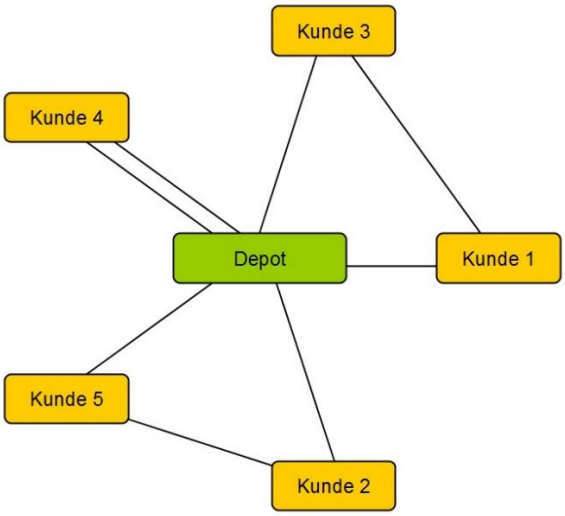


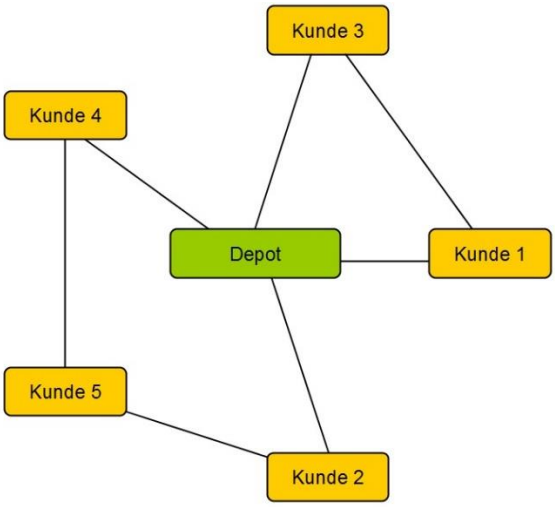
Abbildung 4.2 Pendeltouren werden gebildet



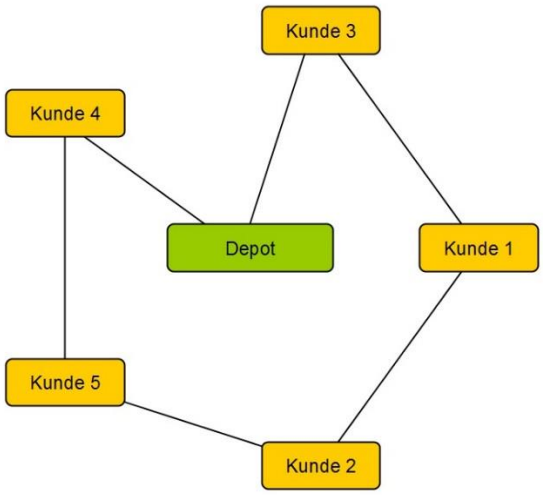
(4.2a) Verbinden der Subtouren …



(4.2b) ...solange keine…



(4.2c) …Regeln verletzt werden



(4.2d) fertige Rundreise

Über die Saving-Heuristik gibt es mittlerweile eine hohe Dichte an wissenschaftlichen Arbeiten, die sich damit befassen, den Algorithmus schneller und effektiver zu machen. Bei einer guten Implementierung kann eine Laufzeit von $\mathcal{O}(n^2 * \log n)$ erreicht werden. Allerdings stößt man bei seriellen Berechnungen schnell an einen hohen Zeitaufwand. Hier setzt diese Arbeit mit hohen parallelisierten Berechnungen im folgenden Kapitel an.

4.2 Parallelisierung der Savings-Heuristik

Wie in Kapitel 4.1 bereits erwähnt wurde, kann die Savings-Heuristik durch parallele Berechnungen zeitlich beschleunigt werden, dabei bleibt die Laufzeit nach der Landau Notation unberührt. Mögliche Parallelisierungen werden in dem folgenden Kapitel näher beschrieben. Der Inhalt von Kapitel 2 und Kapitel 3 wird an dieser Stelle als Wissen vorausgesetzt.

Im ersten Schritt werden alle Entfernungen zwischen allen Knoten berechnet. Da jede einzelne Teilstrecke unabhängig ist, können diese Berechnungen leicht parallelisiert werden. Sei zG die nötige Zeit aller Berechnungen, zE die Berechnungszeit einer einzelnen Strecke von a nach b und n die Anzahl der Knoten im Netzwerk, dann ergibt sich eine theoretische, serielle Berechnungszeit von:

$$zG = zE * (n^2 - n)$$

Diese wird nun auf die Anzahl N der zur Verfügung stehenden Prozessoren verteilt.

$$zG = \frac{zE * (n^2 - n)}{\mathrm{N}}$$

Das Rechnen auf einer GPU kraft mehrerer Tausend Prozessoren kann deutlich schneller sein. Allerdings ist diese Angabe nur theoretisch, da weitere Faktoren, wie zum Beispiel Speicherzugriffszeiten oder die Leistung des Systems, dieses beeinflussen. In Tabelle 4.1 ist ein Vergleich der Entfernungsberechnung, die von unterschiedlich großen Instanzen auf dem Testsystem durchgeführt wurden, zu sehen. Hierbei wurde die benötigte Zeit ohne Speicherreservierung und Kopieren auf die GPU gemessen. Als Formel wurde die Berechnung der Orthodrome zu Grunde gelegt.

Anzahl Knoten im Netzwerk	CPU Intel Core I7 6700K		GPU GTX 1080Ti
	1 Kern	8 Kerne	3584 CUDA-Cores
10 (f1.osm)	0.0153283 ms	12 ms	0.1129188 ms
10317(f2.osm)	5868.4 ms	1578.0 ms	97.4 ms
27514 (f3.osm)	45951.5 ms	11317.0 ms	600.8 ms

Tabelle 4.1 Berechnungszeiten für (n^2-n) Entfernungen auf dem Testsystem

Es fällt schnell auf, dass bei einer sehr kleinen Instanz der Aufwand für parallele Berechnungen die Rechenzeit beeinflussen kann. Dies geschieht unter anderem durch die Verwaltung der einzelnen Threads. Bei den großen Instanzen wird dieses Problem nicht mehr sichtbar, da die parallelisierten Berechnungen dieses kompensieren.

In dem Testprogramm wurden bewusst $(n^2 - n)$ Entfernungen berechnet, damit bereits am Start alle Entfernungen zur Verfügung stehen. Die Berechnung der Pendeltouren folgt demselben Prinzip, allerdings wurde dort bei der Implementierung die Gesamtanzahl der Berechnungen, wie in Tabelle 4.2 zu sehen ist, auf $(n - 1)$ reduziert. Die Berechnung der Pendeltouren ist in einem ungerichteten Netzwerk nicht notwendig, da das Saving auch ohne diesen Wert leicht errechnet werden kann. Sollte allerdings aufgrund von Restriktionen der Hin- und Rückweg unterschiedlich sein, werden zur Erreichung einer Verbesserung die Pendeltouren benötigt.

Anzahl Knoten im Netzwerk	CPU Intel Core I7 6700K		GPU GTX 1080Ti
	1 Kern	8 Kerne	3584 CUDA-Cores
10 (f1.osm)	0.0000000 ms	0.0038319 ms	0.0615446 ms
10317(f2.osm)	0.0208367 ms	0.0179187 ms	0.1029737 ms
27514 (f3.osm)	0.0534386 ms	0.0390697 ms	0.3063861 ms

Tabelle 4.2 Berechnungszeiten für (n-1) Pendeltouren auf dem Testsystem

Bei den Berechnungen der Pendeltouren ist die Instanzgröße deutlich kleiner. Um auf der GPU eine Verbesserung zu erlangen, reicht diese Instanzgröße nicht aus. Jedoch ist zu sehen, dass der Prozessor mit acht Kernen im Vorteil ist. Auch bei den Savings ist eine parallele Berechnung möglich, da auch hier alle benötigten Daten unabhängig voneinander sind. Auf

dem Testsystem, wie in Tabelle 4.3 zu sehen ist, wurden zum Vergleich $(n^2 - n)$ Savings berechnet, die auf $x = \frac{n*(n-1)}{2} - n + 1$ reduziert werden können.

Anzahl Knoten im Netzwerk	CPU Intel Core I7 6700K		GPU GTX 1080Ti
	1 Kern	8 Kerne	3584 CUDA-Cores
10 (f1.osm)	0.0005109 ms	0.0038316 ms	0.0547605 ms
10317(f2.osm)	347.3 ms	395.0 ms	25.3 ms
27514 (f3.osm)	2500.7 ms	5830.9 ms	159.3 ms

Tabelle 4.3 Berechnungszeiten für (n²-n) Savings auf dem Testsystem

Wird die Gesamtlaufzeit in Tabelle 4.4 vom Start des Programms bis einschließlich der Berechnungen der Savings betrachtet, ist eindeutig zu erkennen, dass sich, aufgrund der steigenden Anzahl an Knoten und Berechnungen, der Vorteil der seriellen Programmierung immer mehr hin zur parallelen Berechnung auf der Grafikkarte verschiebt.

Anzahl Knoten im Netzwerk	CPU Intel Core I7 6700K		GPU GTX 1080Ti
	1 Kern	8 Kerne	3584 CUDA-Cores
10 (f1.osm)	0.37 ms	12.62 ms	3410.59 ms
10317(f2.osm)	7031.29 ms	2229.84 ms	2374.94 ms
27514 (f3.osm)	58838.98 ms	19793.11 ms	4900.85 ms

Tabelle 4.4 Gesamtberechnungszeit von Entfernung bis Savings auf dem Testsystem

Dabei verliert die Zeit für das Kopieren der Daten von dem Arbeitsspeicher auf die Grafikkarte und zurück bei steigender Instanzgröße, immer mehr an Bedeutung. Weitere Parallelisierungen sind im Bereich des Sortierens möglich, auf das im folgenden Kapitel näher eingegangen wird.

4.3 Sortieren der Savings

Nach dem in Kapitel 4.2 auf die Parallelisierbarkeit der Berechnungen eingegangen wurde, wird in diesem Kapitel der „Heapsort“-Algorithmus nähergebracht.

Es gibt eine Vielzahl an Sortieralgorithmen, die verwendet werden können. Allerdings muss an dieser Stelle gesagt werden, dass ein direkter Vergleich eines Sortieralgorithmus in serieller und paralleler Version im Rahmen dieser Thesis nicht vorliegt, da es den Rahmen deutlich überschreitet.

Im Testprogramm wurde die serielle Version des Heapsort implementiert. Allerdings wird darauf hingewiesen, dass ein Sortieralgorithmus, wie zum Beispiel der „Radixsort“[12], auf einer Grafikkarte zeitlich deutlich schneller sein kann. Der Heapsort hat im besten und im schlechtesten Fall eine Laufzeit von $\mathcal{O}(n * \log(n))$ und gegenüber anderen Algorithmen, wie zum Beispiel der „Quicksort“ oder „Mergesort“, den Vorteil keinen weiteren Speicher zu benötigen, da dieser bei großen Instanzen eine wichtige Rolle spielt. Aus diesem Grund wurde im Testprogramm der Heapsort implementiert.

Beim Heapsort werden am Anfang alle zu sortierenden Zahlen, wie in Abbildung 4.3 dargestellt, in einer binären Baumstruktur angeordnet. Nun wird überprüft, ob überall die Heapbedingung erfüllt sind. Im ersten Schritt wird bei der Wurzel angefangen und das Kind wird mit dem Elter verglichen. Der Elter muss immer größer sein als die beiden Kinder. Ist dieses nicht der Fall, werden die Positionen vertauscht. Dieser Vorgang wird bei jedem Element wiederholt, bis das größte Element oben steht.

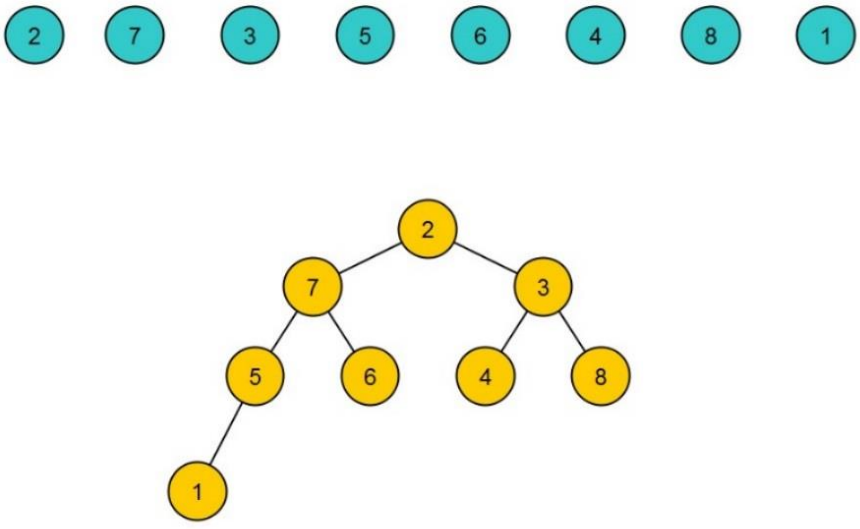
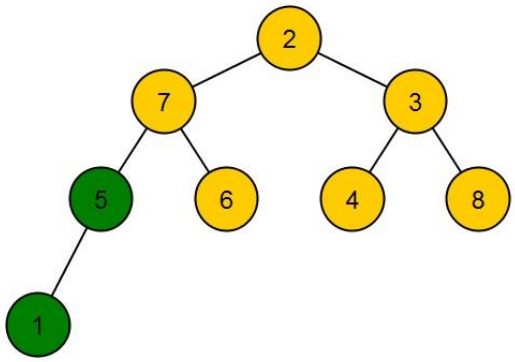
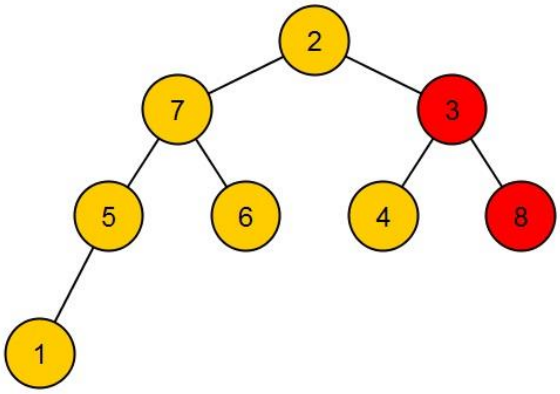


Abbildung 4.3 Blau: Zu sortierenden Zahlen, Gelb: in einer binären Baumstruktur

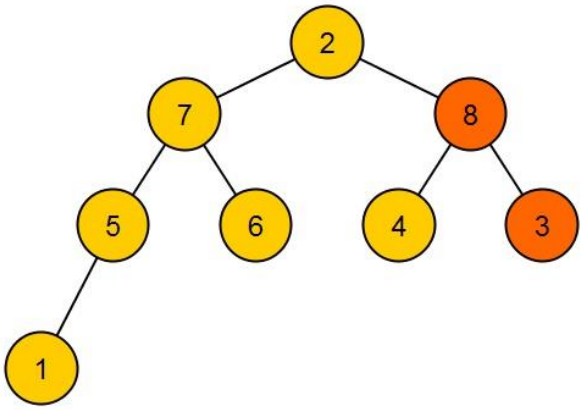
[12] https://en.wikipedia.org/wiki/Radix_sort



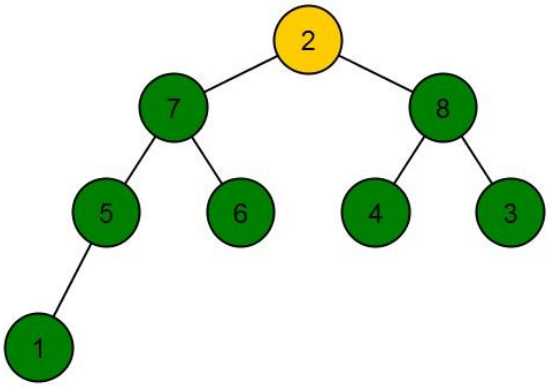
(4.3a) 1 und 5 muss nicht tauschen



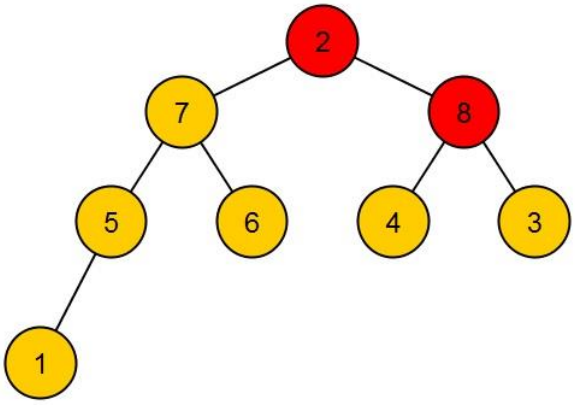
(4.3b) 3 und 8 muss getauscht werden



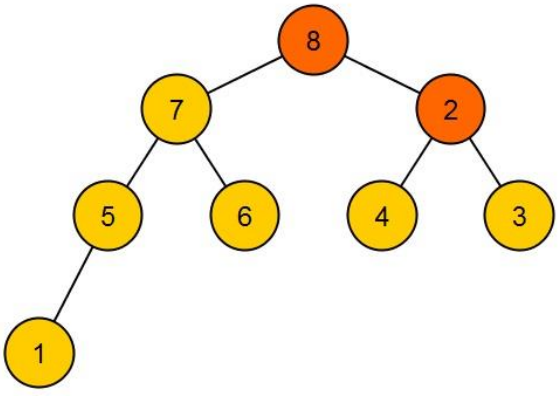
(4.3c) 3 und 8 wurden getauscht



(4.3d) in reihe 1,2 und 3 muss nicht weiter getauscht werden

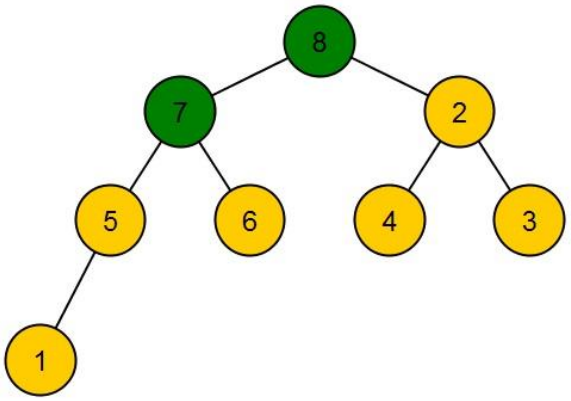


(4.3e) in Reihe 4 muss die 2 mit der 8 aus Reihe 3 getauscht werden

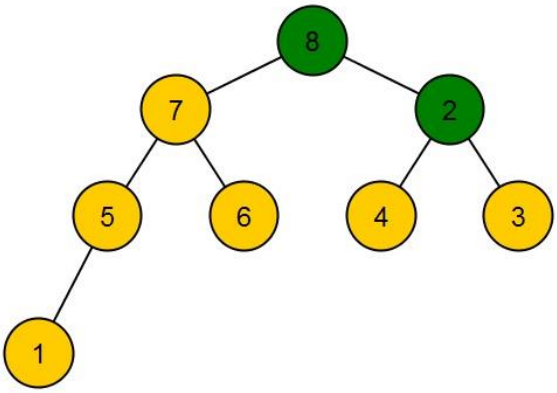


(4.3f) 2 und 8 getauscht die 8 steht nun ganz oben

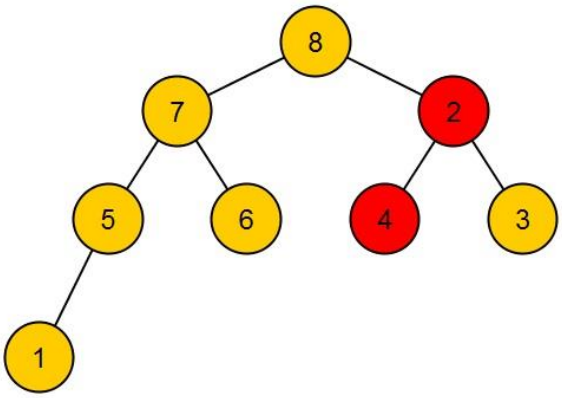
Im zweiten Schritt, dem Versickern, wird von oben angefangen, zu überprüfen, ob die Heapbedingungen auch aus dieser Richtung erfüllt sind.



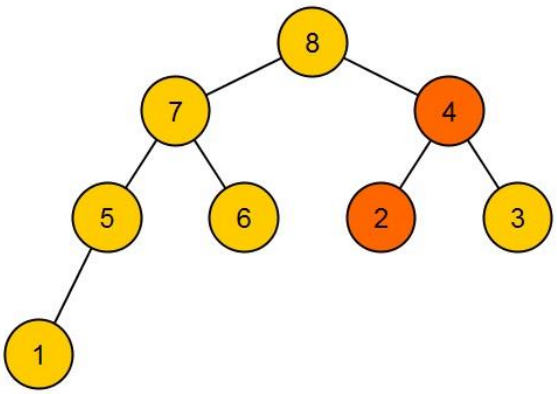
(4.3g) Versickern: 8 ist größer als 7



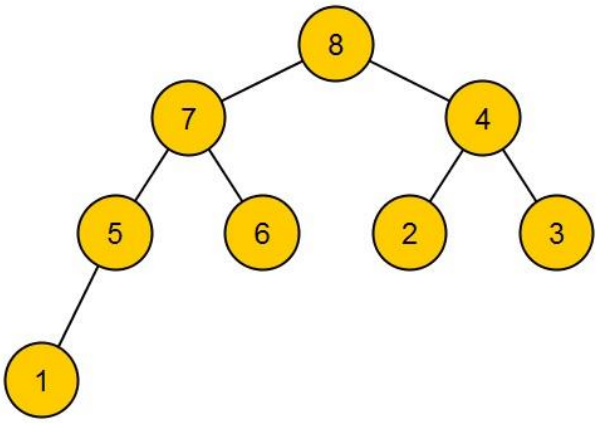
(4.3h) 8 ist größer als 2



(4.3i) 2 ist nicht größer als 4 und muss getauscht werden

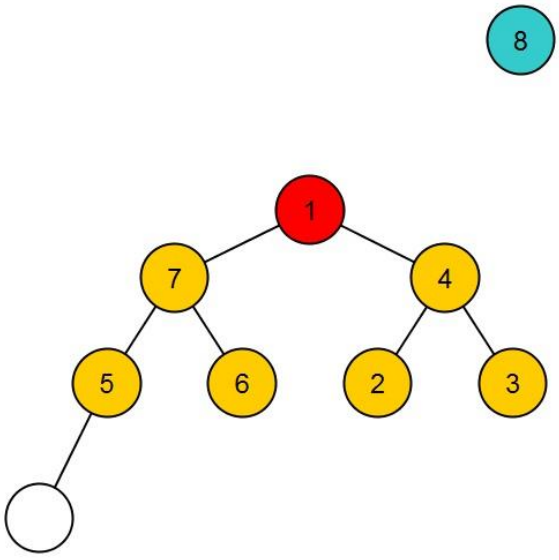


(4.3j) 2 und 4 vertauscht

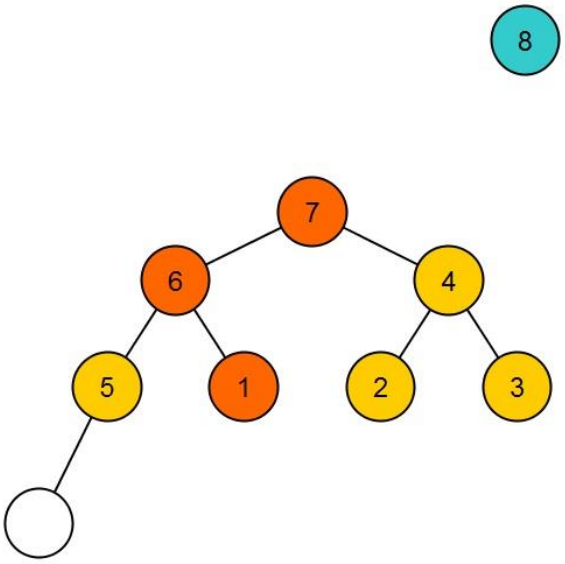


(4.3k) der Baum erfüllt nun die Hauptbedingungen

Nach dem Versickern sind alle Heapbedingungen erfüllt, wodurch ein Maxheap entstanden ist, in dem die größte Zahl ganz oben steht und herausgenommen werden kann. Den Platz an der Spitze des Baumes übernimmt die letzte Zahl, in diesem Fall die 1.

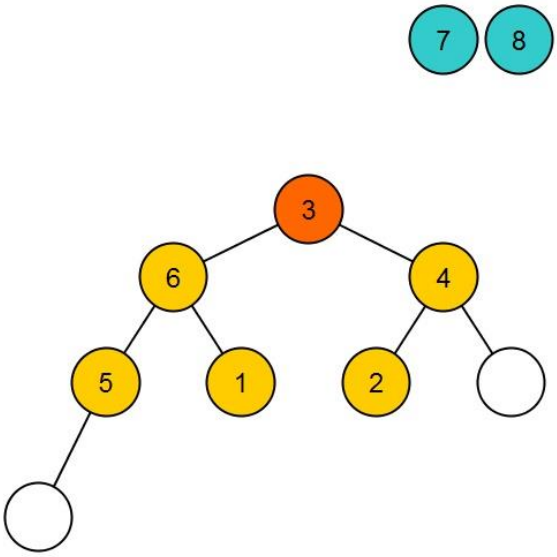


(4.3l) die 8 wurde herausgenommen und die 1 nach oben gestellt

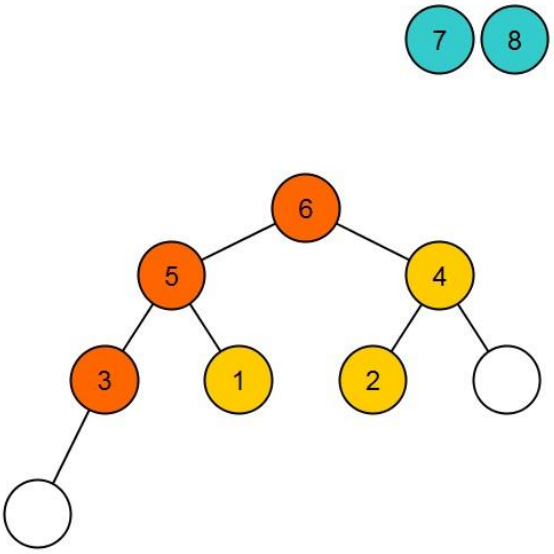


(4.3m) die 1 wird erneut versickert und immer mit dem größten Kind, wenn nötig, vertauscht

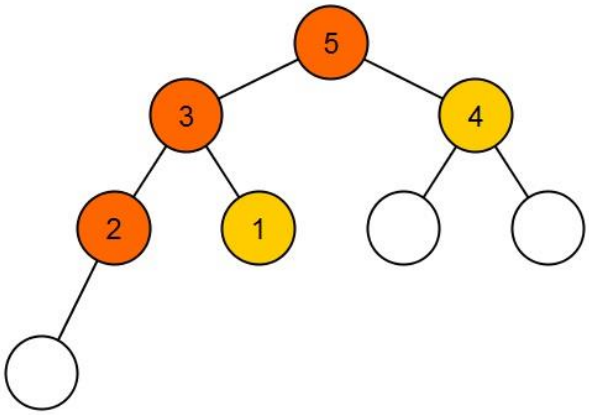
Nun steht wiederum die größte Zahl, hier die 7, oben, die erneut herausgenommen wird. Dieser Vorgang wird solange wiederholt, bis alle Elemente abgearbeitet sind und die Zahlen aufsteigend sortiert vorliegen.



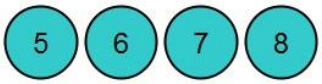
(4.3n) 7 herausgenommen und 3 nach vorne gestellt

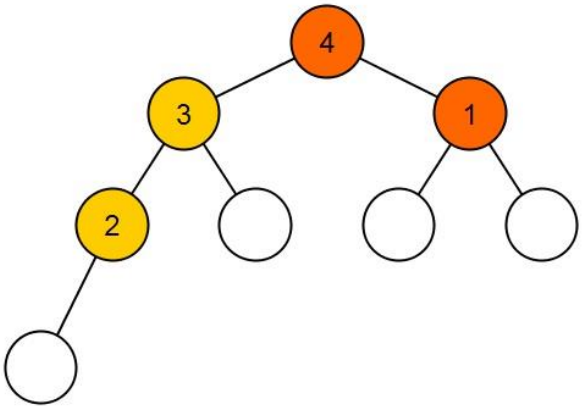


(4.3o) 3 wird versickert

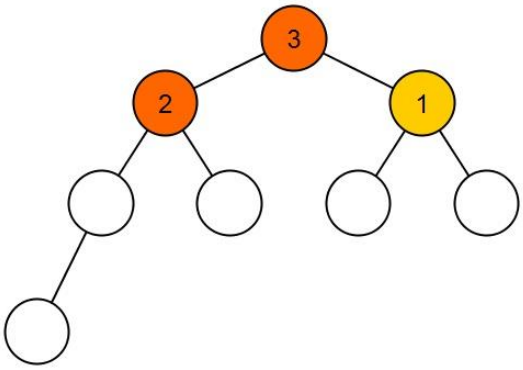


(4.3p) 6 herausgenommen, 2 nach vorne gestellt und versickert

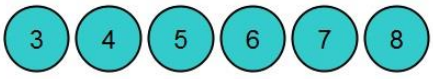


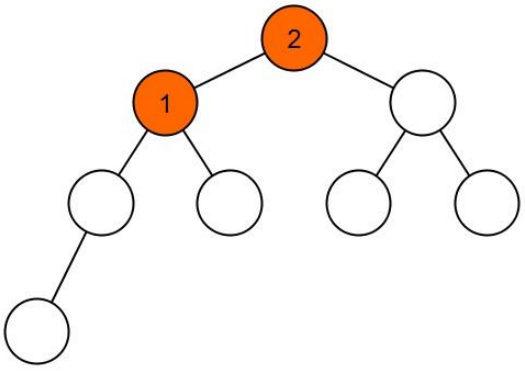


(4.3q) 5 herausgenommen, 1 nach vorne gestellt und versickert

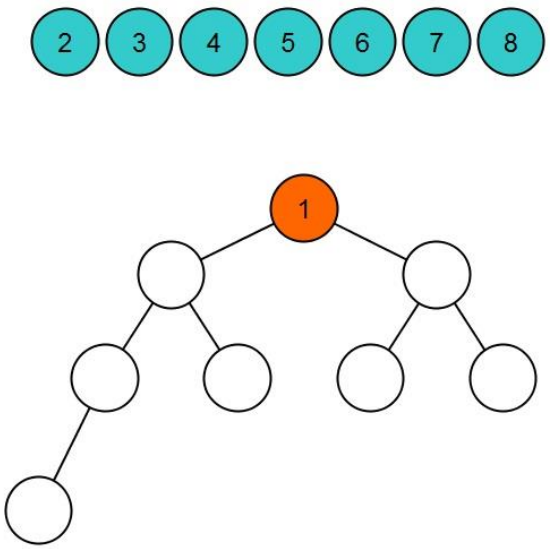


(4.3r) 4 herausgenommen, 2 nach vorne gestellt und versickert

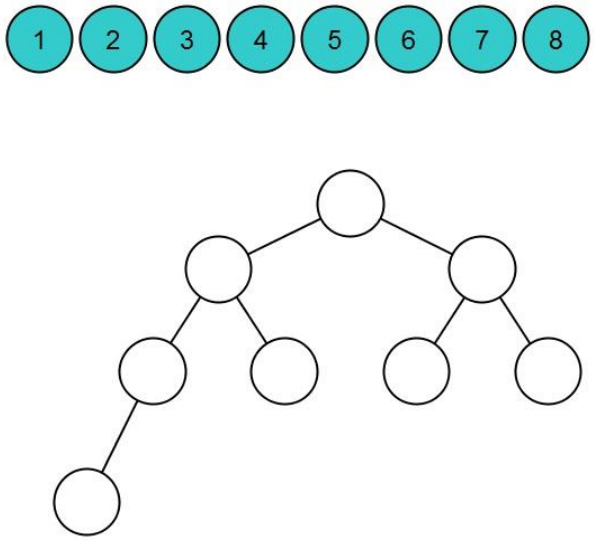




(4.3s) 3 herausgenommen, 1 nach vorne gestellt und versickert



(4.3t) 2 herausgenommen, 1 nach vorne gestellt und versickert



(4.3u) der Baum ist nun leer und alle Zahlen sind aufsteigend sortiert

Wie bereits erwähnt, kommt der Heapsort ohne nennenswerten zusätzlichen Speicher aus. Näheres zur Implementierung im Testprogramm wird im Kapitel 5 erläutert.

5. Praktischer Teil

In Kapitel 4 wurden, unter anderem, Anwendungsmöglichkeiten in der Logistik von parallelen Berechnungen mit Hilfe der Savings-Heuristik erörtert, während sich dieses Kapitel mit der Implementierung in der Programmiersprache C befasst. Dabei wird auf spezielle Probleme und deren Lösungen eingegangen. Grundkenntnisse der Informatik werden an dieser Stelle vorausgesetzt. Detaillierte Informationen können dem Quellcode in der Anlage entnommen werden. In Kapitel 5.1 wird der allgemeine Aufbau des Programmes erläutert.

5.1 Allgemeiner Aufbau des Programmes

Wie zuvor bereits erwähnt wird in diesem Kapitel näher auf den Aufbau eingegangen. Das Programm wurde mit Microsoft Visual Studio 2013 inklusive der CUDA 9.0 Runtime Bibliothek entwickelt. Sämtliche Projektdateien befinden sich auf der CD in der Anlage, in dem Ordner „Vergleich“. Da es sich hierbei um ein CUDA® Projekt handelt, hat die Hauptprogrammdatei nicht die übliche Dateiendung „.c“, sondern „.cu“ und befindet sich im Ordner „Vergleich/Vergleich“. Die „Kernel.cu“ kann wie gewohnt oder alternativ auch mit einem normalen Texteditor, wie Notepad, geöffnet werden. Das Programm teilt sich in vier Hauptbereiche.

- Abfragen und Einlesen der Daten
- Berechnungen über CPU
- Berechnungen über CPU mit mehreren Kernen
- Berechnungen über die GPU

Beim Einlesen der Daten wird als erstes die ausgewählte .osm Datei gelesen und die Anzahl der Knoten ermittelt. Im nächsten Schritt kann der Nutzer wählen, welche Berechnungen durchgeführt werden sollen. Dabei kann er verschiedene Varianten zwischen CPU, MCPU[13] und GPU kombinieren. Als nächstes hat der Nutzer die Möglichkeit detailliertere Informationen während der Berechnungen einzuschalten. Hierbei ist zu erwähnen, dass bei einer Instanz von mehr als zehn Knoten diese Option nicht empfohlen wird, da einige Informationen aufgrund der Größe unübersichtlich dargestellt werden können. Desweiteren können Berechnungszeiten durch zusätzliche Bildschirmausgaben verfälscht werden, denn der Computer muss auch diese Befehle verarbeiten. Als weitere Auswahlmöglichkeit wird der Nutzer nach dem Depot gefragt. Hier kann entschieden werden, der wievielte eingelesene Knoten das Depot ist. An dieser Stelle wird empfohlen, die Rohdaten in der .osm Datei im Vorfeld zu bearbeiten und den Knoten vom Depot an die erste Stelle zu kopieren,

[13] MCPU = Multi Central Processing Unit, Ein Prozessor mit mehreren Kernen wird im Quellcode MCPU genannt

um anschließend mit Knoten eins fortfahren zu können. Sollten die detaillierten Informationen ausgewählt worden sein, kommt nun eine Abfrage für zwei weitere Knoten (Flughäfen). Diese werden zusätzlich zum Depot mit Entfernung und Savings ausgegeben. Sind alle Eingaben getätigt, wird anhand der zuvor ermittelten Knotenanzahl im nächsten Schritt der benötigte Arbeitsspeicher berechnet. Sollte der zur Verfügung stehende Speicher zu gering sein, bekommt der Nutzer die Möglichkeit abzubrechen oder fortzufahren. Ein Fortfahren an dieser Stelle kann zu Fehlern und Abstürzen führen. Ist zuvor die Berechnung über die GPU ausgewählt worden, erfolgt als letzte Abfrage eine Auswahl der Sortieralgorithmen. Hier wird deutlich auf den Heapsort verwiesen. Der zur Auswahl stehende „Odd-Even-Sort“[14] ist bereits für die parallele Programmierung auf der Grafikkarte implementiert, ist aber wie bereits in Kapitel 4.3 erwähnt, nicht Gegenstand dieser Arbeit, da kein direkter Vergleich in der Berechnungszeit zum seriellen Heapsort stattfindet. Weiterhin wird an dieser Stelle im Hintergrund die Grafikkarte auf CUDA® Fähigkeit überprüft. Sollte dieses nicht gegeben sein, wird die Berechnung auf der Grafikkarte automatisch deaktiviert. Nachdem alle Eingaben und Prüfungen durchgeführt wurden, reserviert das Programm den notwendigen Speicher, um im Anschluss alle Daten aus der .osm Datei einzulesen.

Nun folgen die gewählten Berechnungen. Je nach Auswahl werden nacheinander CPU, MCPU und zum Schluss die GPU Berechnung durchgeführt. Dabei laufen alle drei nach dem folgenden Schema ab.

- Entfernungsberechnung
- Pendeltouren berechnen
- Savings berechnen
- Sortieren der Savings
- Überprüfung der Sortierung
- Touren zusammenstellen
- Ausgabe in eine .osm Datei

Dabei unterscheiden sich alle drei Versionen in ihrer Programmierung. Während bei der CPU auf eine serielle „Standard“ C Programmierung zurückgegriffen worden ist, wurde bei der MCPU Version auf die Programmierschnittstelle „openMP“[15] zurückgegriffen und mit Hilfe von „#pragma“[16] Kompelier-Derektiven die Parallelität ermöglicht. Die Berechnungen auf der Grafikkarte werden durch die CUDA® Programmierschnittstelle gewährleistet. Wurde am Anfang die Ausgabe von Details gewählt, werden am Ende des Programmes einige Zusatzinformationen ausgegeben.

[14] https://en.wikipedia.org/wiki/Odd%E2%80%93even_sort
[15] http://www.openmp.org/
[16] https://gcc.gnu.org/onlinedocs/cpp/Pragmas.html

Nach dem in diesem Kapitel der allgemeine Ablauf des Programmes beschrieben wurde, wird in den folgenden Kapiteln detaillierter auf einzelne Bereiche im Programm eingegangen.

5.2 Ausgangsdaten, die Knoten Id und GPS Koordinaten

Im vorherigen Kapitel wurde der allgemeine Ablauf des Programmes beschrieben. In diesem Kapitel wird näher auf die Verwendung und Bearbeitung einzelner Dateien eingegangen.

Die Knoten Id´s sind in der XML Datei von Openstreetmap als Ganzzahl hinterlegt. Hierfür wurde als erstes der Datentyp integer überprüft. Beim „signed integer“ ist die größte mögliche Zahl, die gespeichert werden kann, 2.147.483.647. Die Knoten Id´s von Openstreetmap können Werte von über $4.2*10^9$ erreichen, daher ist dieser Datentyp nicht geeignet. Eine Alternative wäre „unsigned integer“, aber selbst dieser Datentyp hat ein Limit bei rund $4.3*10^9$ und kann daher nicht sicherstellen, dass alle Werte gespeichert werden. Die Knoten Id von Openstreetmap ist eine interne, fortlaufende Nummer. Wird diese nicht benötigt, kann bei der Implementierung auf solche großen Zahlen verzichtet und eigene Zahlen vergeben werden. Für das Testprogramm wurde als dritte Möglichkeit der Datentyp „float“ gewählt, da hier die größte mögliche Zahl bei rund $3.4*10^{38}$ liegt und sie aufgrund dieser Dimension gut geeignet ist, die originalen Knoten Id´s zu speichern. Allerdings kann es aufgrund der Datentypen zu Ungenauigkeiten[17] kommen. Die Knoten Id wird in diesem Programm nicht weiter verwendet, daher wird sie bei der Ausgabe der .osm Datei durch eine eigene Nummerierung ersetzt, um eine bessere Übersichtlichkeit zu gewähren.

Die GPS Koordinaten sind als positive oder negative Gleitkommazahl mit bis zu sieben Nachkommastellen hinterlegt, aus diesem Grund wurden die Datentypen „float“ und „double“ überprüft. Beide Typen sind geeignet, allerdings benötigt eine Zahl vom Typ double eine Speichergröße von 64 Bit und ist durch die 15 Nachkommastellen deutlich zu groß dimensioniert. Desweiteren unterstützt das in Kapitel 4 vorgestellte CUDA® nur 32 Bit Gleitkommazahlen vom Typ float. Der Datentyp double kommt für diese Aufgabenstellung dadurch nicht mehr in Frage. Der Übersichtlichkeit halber wurde sowohl der Längengrad als auch der Breitengrad in einem einzelnen Vektor, $h\_FloatLat$ und $h\_FloatLon$, vom Typ float gespeichert. Alternativ kann an dieser Stelle auch eine zweidimensionale Matrix verwendet werden. Bei der Nutzung von CUDA® ist zu empfehlen, die Namen von Variablen so zu kennzeichnen, dass zu erkennen ist, ob sie auf dem Host oder auf dem Device zu finden sind. Im Testprogramm wurde dies durch das Voranstellen eines $h\_$ für Hostvariablen und $d\_$ für Devicevariablen gelöst. Es ist möglich denselben Namen sowohl auf dem Host als auch auf dem Device zu nutzen, was allerdings die Fehlersuche erheblich erschweren kann.

[17] Jürgen Wolf, C von A bis Z, Galileo Computing

5.3 Die globale Id

Damit die Berechnungen immer mit den maximal zur Verfügung stehenden CUDA-Cores durchgeführt werden können, darf die Anzahl der Threads nicht schon im Vorfeld durch andere Funktionen oder ähnliches verkleinert werden. Desweiteren muss eine Möglichkeit gegeben sein, in der Daten effektiv gespeichert werden können.

Eine einfache Entfernungsmatrix für ein gerichtetes Netzwerk in zweidimensionaler Form würde eine Größe von n^2 benötigen. Für die Entfernungsberechnung würden in serieller Programmierung zwei geschachtelte Schleifen ausreichen. Damit diese Matrix auf der GPU berechnet werden kann, ist ein erhöhter Programmieraufwand von Nöten. Ungeachtet dessen, ist sowohl bei der seriellen als auch bei der parallelen Implementierung, bei Berechnungen einer zweidimensionalen Matrix, der x und y Wert für eine Berechnung notwendig. Diese werden meist mittels einer Funktion übergeben. Eine einfache Implementierung in CUDA® wäre eine Schleife der Größe n zu starten, die jeweils ein Kernel mit n Berechnungen einleitet. Allerdings kann dies dazu führen, dass bei bestimmter Instanzgröße nicht die volle Leistung der Grafikkarte genutzt wird. Ein weiteres Kontra für eine zweidimensionale Matrix im Rahmen dieser Fragestellung ist das Sortieren der Savings aus Kapitel 4.3. Hierfür müsste ein zusätzlicher Speicher der Größe $3n$ den x und y Wert sowie das Saving zum Sortieren aufnehmen. Aus den genannten Gründen ist eine zweidimensionale Speicherung bedingt geeignet.

Eine effektivere Lösung wurde bereits in Kapitel 3.2 erwähnt, der globale Thread Index von CUDA®. Angelehnt an diesen wurde für diese Bachelorarbeit eine denkbare Implementierung in einem eindimensionalen Vektor genutzt.

Als erstes wurde, wie in Tabelle 5.1 dargestellt, eine globale Id festgelegt. Dabei entspricht die Id 0 der Verbindung von Knoten 1 zu Knoten 1, die Id 1 der Verbindung von Knoten 1 zu Knoten 2 und so weiter. Daraus ergibt sich eine gesamte Anzahl von n^2 globalen Id´s. Dies entspricht der Größe der zu vor erwähnten, zweidimensionalen Matrix, hat aber den Vorteil, dass nur noch die globale Id z an eine Funktion übergeben werden muss und zum Sortieren nur noch ein Speicher der Größe n benötigt wird. Näheres dazu wird in Kapitel 5.3 erläutert.

	1	2	3	4	5	6	7	8	9	10
1	0	1	2	3	4	5	6	7	8	9
2	10	11	12	13	14	15	16	17	18	19
3	20	21	22	23	24	25	26	27	28	29
4	30	31	32	33	34	35	36	37	38	39
5	40	41	42	43	44	45	46	47	48	49
6	50	51	52	53	54	55	56	57	58	59
7	60	61	62	63	64	65	66	67	68	69
8	70	71	72	73	74	75	76	77	78	79
9	80	81	82	83	84	85	86	87	88	89
10	90	91	92	93	94	95	96	97	98	99

Tabelle 5.1 beispielhafte globale Id´s (schwarz) bei n=10(rot)

Die globale Id der Verbindung von x nach y lässt sich über die Formel

$$globalId = (x-1) * n + (j-1)$$

errechnen. Umgekehrt wird der x und y Wert aus der globalen Id wie folgt berechnet

$$x = \left\lfloor \frac{globaleId}{n} \right\rfloor + 1$$
$$y = (globaleId \ mod \ n) + 1$$

Wird für die globale Id der Datentyp integer genutzt, ist bei x ein Abrunden nicht nötig, da dies bereits durch die ganzzahlige Division geschieht.

5.4 Entfernungsberechnung

Die Entfernung beschreibt die Strecke zwischen zwei Knoten, hier A und B genannt. Unter Verwendung von GPS Koordinaten lässt sich die reale Entfernung, die Orthodrome, auf der Erde berechnen. Hierbei wird auf die Berechnung des Großkreisbogens zurückgegriffen.

Sei ϕ der geografische Breitengrad und λ der geografische Längengrad im Bogenmaß, dann errechnet sich die Entfernung $L \ in \ km$ zwischen den beiden Knoten durch

$$L = (\text{acos}\left(sin(\phi_A)sin(\phi_B) + cos(\phi_A)cos(\phi_B)cos(\lambda_B - \lambda_A)\right) * Erdradius \ in \ km$$

Die Programmiersprache C stellt in der „*math.h*“ Bibliothek die Sinus- und Cosinus-Funktion zur Verfügung. Allerdings sind, unter anderem, die sin(), cos() und acos() Funktionen auf den Datentyp double beschränkt. Ihnen muss somit ein double Wert übergeben werden, als Rückgabewert erhält man auch einen double Wert.

Da dieses, wie bereits erwähnt, nicht von CUDA® unterstützt wird, muss für die Berechnung auf der Grafikkarte an dieser Stelle auf die float Version sinf(), cosf() und acosf() zurückgegriffen werden. Die serielle Programmierung auf der CPU bleibt hiervon unberührt, allerdings wird zum zeitlichen Vergleich auch hier die float Version genutzt.

Zur Bereitstellung des für die Berechnung benötigten Bogenmaßes ζ, wurden am Anfang der Entfernungsberechnung alle Längen- und Breitengrade, α in Grad, mittels der Formel

$$\zeta = \frac{2\pi * \alpha^\circ}{360^\circ}$$

umgerechnet.

Die GPS Koordinaten bieten mit der fünften Dezimalstelle eine Genauigkeit von bis zu einem Meter. Diese Genauigkeit sollte im Testprogramm auch bei den Entfernungen beibehalten werden. Daher empfiehlt sich hier die Gleitkommazahl zu nutzen. Für das Testprogramm wurde auch hier, zur Sicherung der Entfernung in Kilometern, auf den Typ float zurückgegriffen.

5.5 Savings und Sortierung

Zuerst muss der Datentyp zur Speicherung gewählt werden. Dazu wird die Entfernungsberechnung herangezogen. Da diese bereits in Gleitkommazahl vom Typ float vorliegt, kann auch auf diesen Datentypen zurückgegriffen werden. Allerdings ist zu prüfen, welche Genauigkeit von Nöten ist. Reichen bereits ganzzahlige Kilometer, kann unter Umständen auch der Datentyp integer genutzt werden. Für eine hohe Genauigkeit der Savings wird im Testprogramm der Typ float verwendet.

Im zweiten Schritt wird die Art der Speicherung näher betrachtet. Da die Savings in der Anzahl m sortiert werden müssen und die dazugehörigen Knoten ebenfalls abgespeichert werden, war die erste Überlegung, die Daten in einer $3 * m$ großen Matrix, wie in Tabelle 5.2, zu speichern. Dabei wird in den ersten zwei Zeilen der Knoten Index der beiden verwendeten Knoten gespeichert und in der dritten das berechnete Saving.

von	1	1	1	1	1	1	1	1	1	...
nach	2	3	4	5	6	7	8	9	10	...
Savings	37	21	98	11	47	39	1	48	67	...

Tabelle 5.2 beispielhafte Matrix zum Speichern von unsortierten Savings

Als Sortierkriterium kann nun das Saving genommen werden und wenn nötig, wird die ganze Spalte getauscht. Zum Tauschen zweier unterschiedlicher Werte in den Variablen a und b, sind in der Programmiersprache C mindestens drei Schritte zu tätigen.

1. Speichere Wert a temporär in Variable c zwischen
2. Setze $a = b$
3. Schreibe Wert c in Variable b

Dies führt dazu, dass bei sechs zu tauschenden Werten mindestens neun Schritte nötig sind. Desweiteren müssen beim Speichern die Savings von demselben Datentyp wie der Laufindex der Knoten sein, ansonsten kann ein Datenverlust durch Typenumwandlung nicht vermieden werden.

Da der Speicherbedarf bei dieser Lösung immer ein Dreifaches der Anzahl der Savings beträgt, wurde dies als ungeeignet für das Testprogramm eingestuft. Eine effektivere Lösung wurde durch das Verwenden der globalen Id aus Kapitel 5.3 erreicht. Da bereits alle Savings in einem eindimensionalen Vektor, im Testprogramm $s\_savings$, berechnet zur Verfügung stehen und als Laufindex die globale Id genutzt wird, kann zum Sortieren ein Vektor der Größe m genutzt werden. In diesem Vektor werden nur die globalen Id´s gespeichert. Zum Sortieren wird, wie in Abbildung 5.1, über die globale Id auf den Vektor mit den Savings zurückgegriffen.

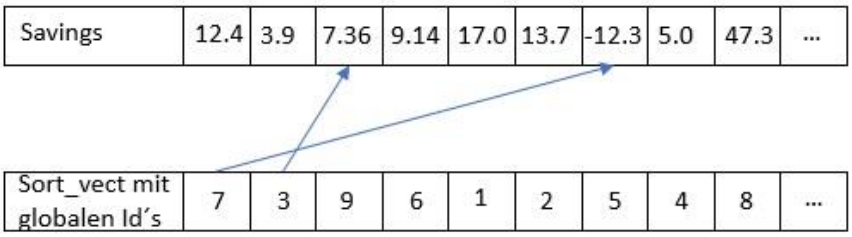


Abbildung 5.1 Verknüpfung der beiden Vektoren über die globale Id

Der Vorteil hierbei ist, dass nur noch ein Wert, in diesem Fall die Reihenfolge der globalen Id, getauscht werden muss und die Savings einen anderen Datentyp haben können. Eine weitere Speicherreduzierung könnte durch das Verkleinern des zu sortierenden Vektors erreicht werden, in dem dieser nur noch die Größe der Anzahl aller positiven Savings hat und auch diese beinhaltet.

5.6 Tour Zusammenstellung

Für die Tour-Zusammenstellung gibt es verschiedene Möglichkeiten, diese hängt in erster Linie von den Restriktionen ab. Da es für diese Thesis keine Restriktionen, wie zum Beispiel maximale Beladung oder maximale Flugzeit gibt, wird versucht, so viele Kunden wie möglich aufgrundlage der Savings in einer Tour zusammenzufassen.

Dafür wurden als erstes zwei integer Vektoren angelegt. Der erste speichert die Anzahl der Kanten, welche zwischen dem Kunden und Depot verlaufen. Diese beträgt durch die Pendeltouren am Anfang eine Größe von zwei. Im zweiten Vektor wird mit Hilfe einer Schleife jedem Kunden eine Subtour Id und dem Depot die Subtour Id 0 zugewiesen.

Im nächsten Schritt können nun alle Subtouren nach den in Kapitel 4.1 angegebenen Regeln überprüft und zusammengeführt werden. Für zwei verschiedene Subtouren wird zur Verbindung im Testprogramm die Subtour Id zur Hilfe genommen. Es wird dabei überprüft, welche der beiden Id´s den größeren Wert hat und im Anschluss durch die Id mit dem kleineren Wert ersetzt.

Nach dem dies durchlaufen wurde, kann es durch negative Savings dazu kommen, dass noch Kunden über zwei Kanten zum Depot verfügen, die nicht in der Subtour 0 eingebunden sind. Daher wird der Vektor der Kantenanzahl daraufhin überprüft, ob noch ein Kunden über zwei Kanten verfügt, wenn ja, werden diese in dem Tourenvektor als negativer Wert gespeichert. Durch diese Kennzeichnung kann bei der Ausgabe der Tour zwischen eigener einzelner Pendeltour und der Haupttour unterschieden werden.

5.7 Exkurs in die Visualisierung

In den vorangegangenen Kapiteln wurde detailliert auf den Ablauf des Programmes eingegangen. Nach dem dieses eine .osm Datei als Ausgabe erstellt hat, wird in diesem Kapitel kurz eine Visualisierungsmöglichkeit dargestellt. Hierbei wird erneut auf die vorangegangene Studienarbeit „Bestimmung optimaler Rundreisen mit realen Openstreetmap-Daten und Visualisierung im Geoinformationssystem QGIS“ verwiesen. Die Nutzung von QGIS mit Openstreetmap sollte an dieser Stelle bekannt sein.

Als Grundlage für dieses Kapitel wurden für die Berechnungen zwei Instanzen verwendet. Die „deu.osm“ beinhaltet GPS Koordinaten von 19 und die „deu2.osm“ von 112 Flughäfen. Alle dazugehörigen Dateien sind auf der CD in der Anlage[18] hinterlegt. Abbildung 5.2 zeigt den Kartenausschnitt in QGIS mit den 19 Flughäfen und dem zentral liegenden Depot in der Nähe von Bückeburg (gelb markiert) aus der Datei deu.osm.

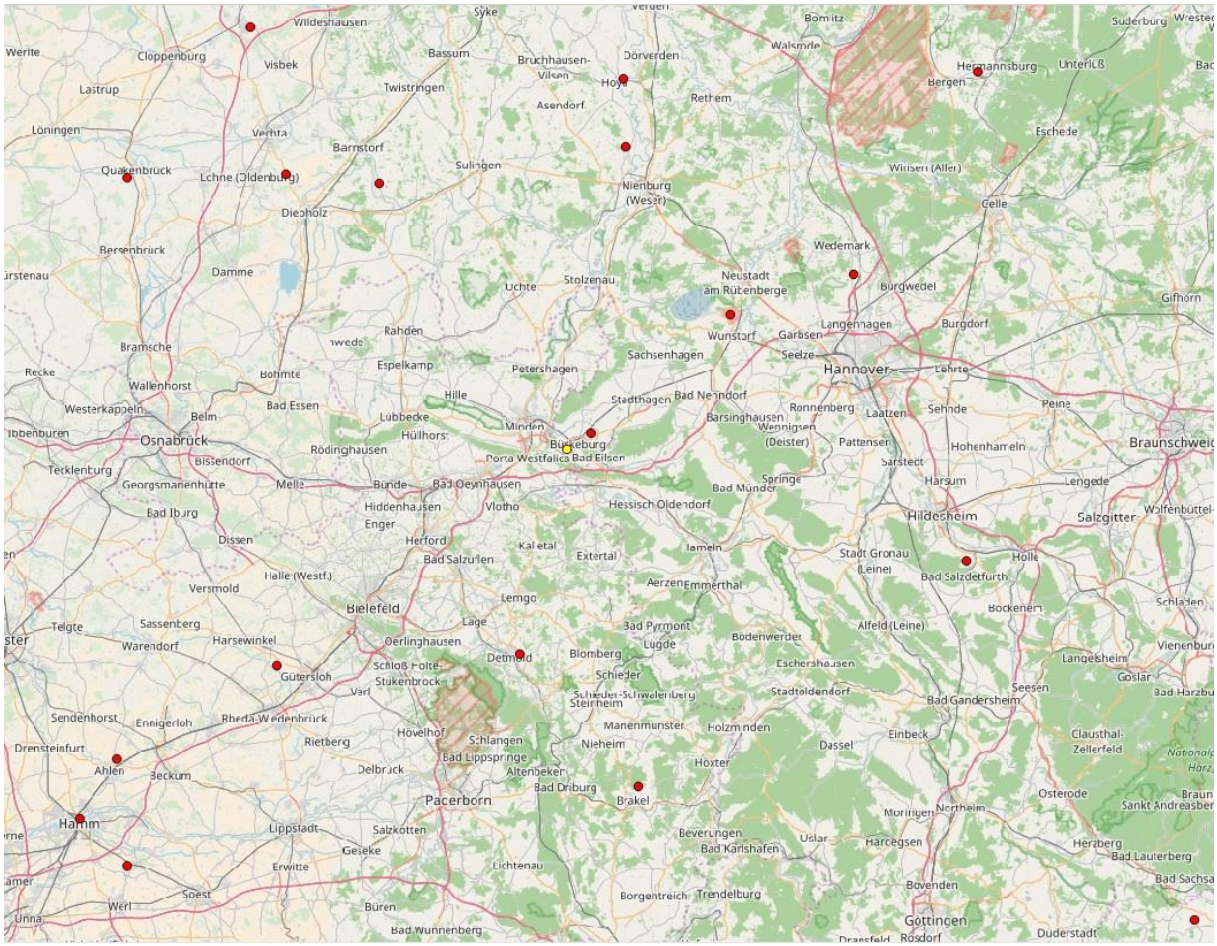


Abbildung 5.2 19 Flughäfen von deu.osm in QGIS dargestellt

[18] CD Ordner: /Vergleich/Beispiel/deu und /Vergleich/Beispiel/deu2

Alternativ zu dieser Abbildung kann mit der „deu.qgs“ Datei die Instanz in QGIS direkt betrachtet oder die „deu1-A.jpg“ hinzugezogen werden. Um die Berechnung durchführen zu können, muss sich die Datei deu.osm in demselben Ordner befinden wie die „Vergleich.exe“. Ist dieses sichergestellt, kann das Programm mit

$$"vergleich.exe\ deu.osm"$$

gestartet werden. Nachdem die Abfragen, wie in Kapitel 5.1 beschrieben, gemacht wurden, werden alle Berechnungen durchgeführt und die entsprechenden Dateien angelegt. Diese können nun, wie in Abbildung 5.3 zusehen ist, als eigenes Layer in QGIS geladen und dargestellt werden.

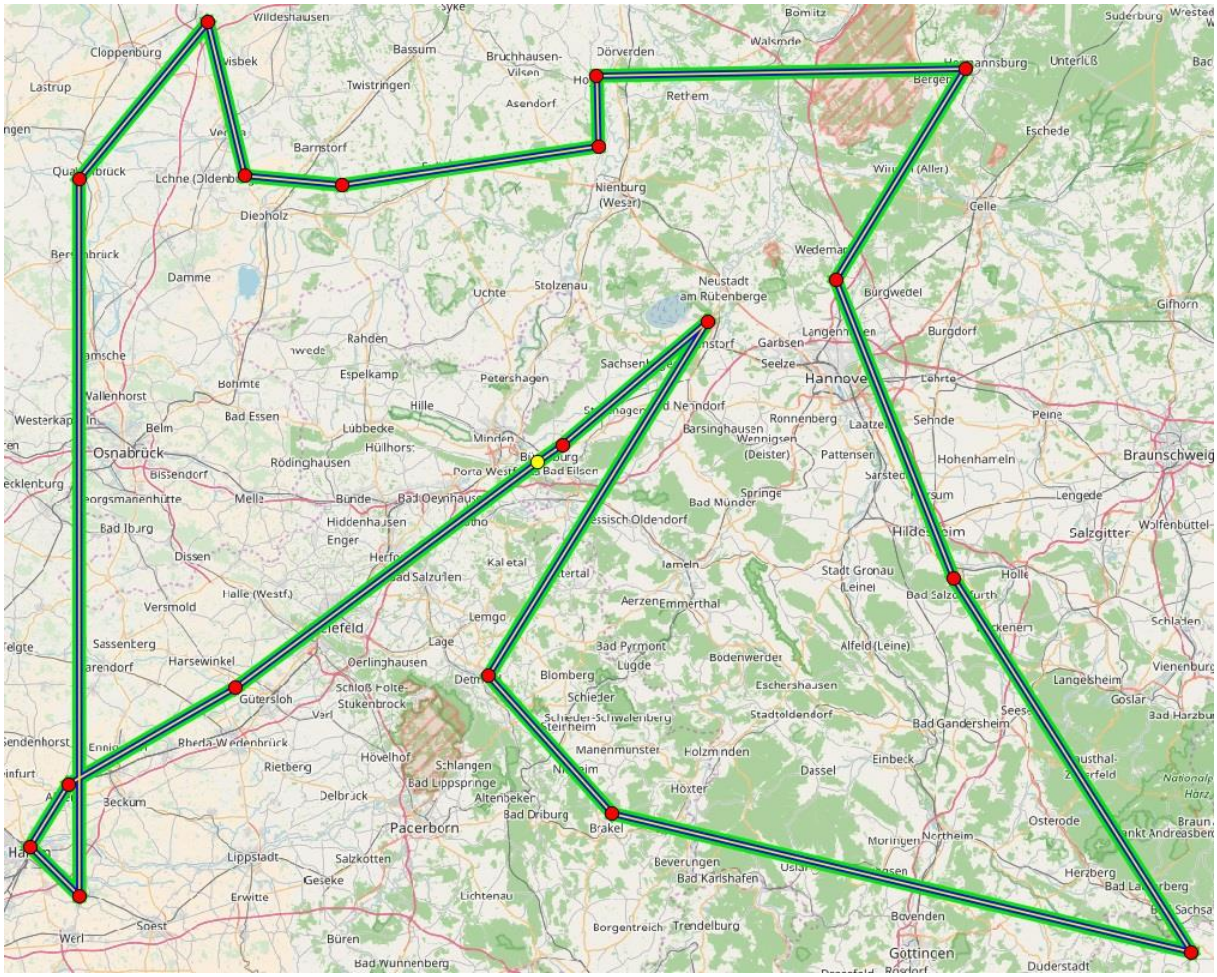


Abbildung 5.3 in QGIS Visualisiertes Ergebnis, mit den drei Layern für CPU(grün), MCPU(blau) und GPU (gelb) übereinandergelegt.

Nach diesem Prinzip wurde auch die zweite Instanz deu2.osm berechnet und wie in Abbildung 5.4 und Abbildung 5.5 visualisiert. Auch hier kann alternativ die „deu2.qgs“ in QGIS betrachtet werden.

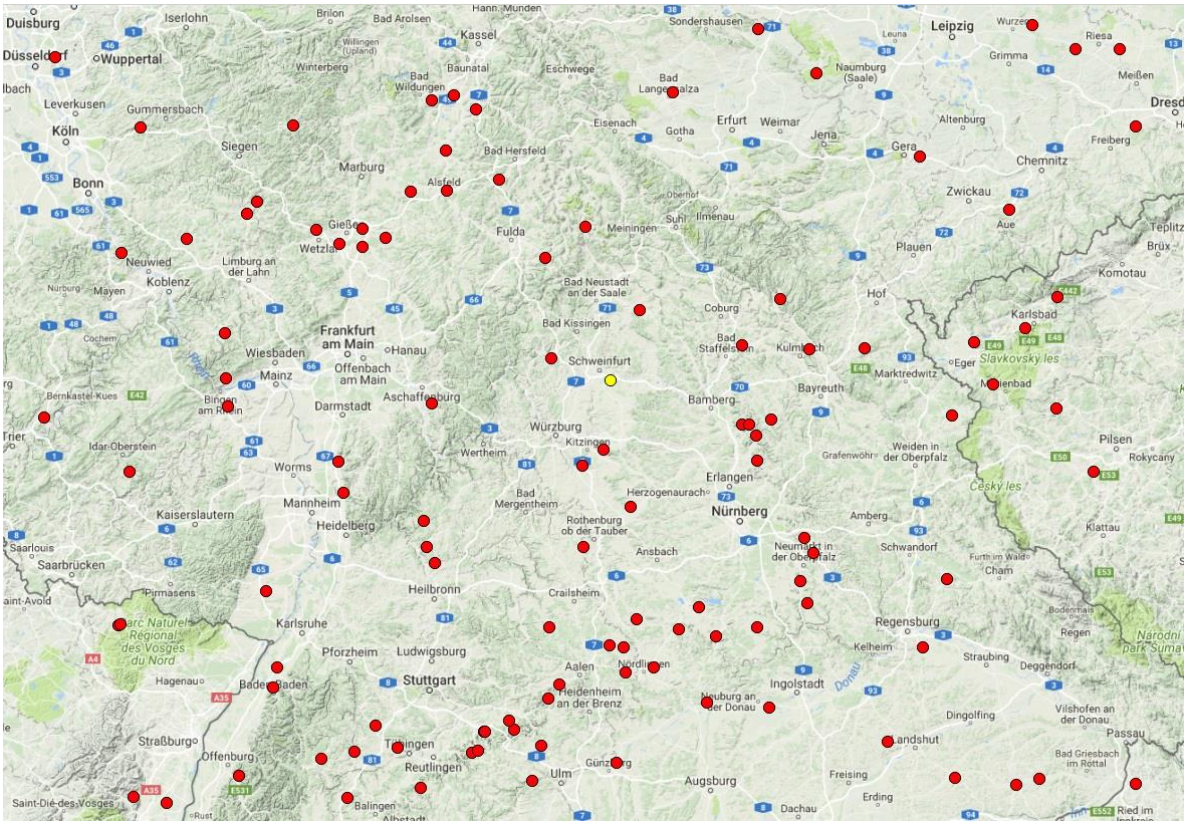


Abbildung 5.4 112 Flughäfen von deu2.osm in QGIS dargestellt

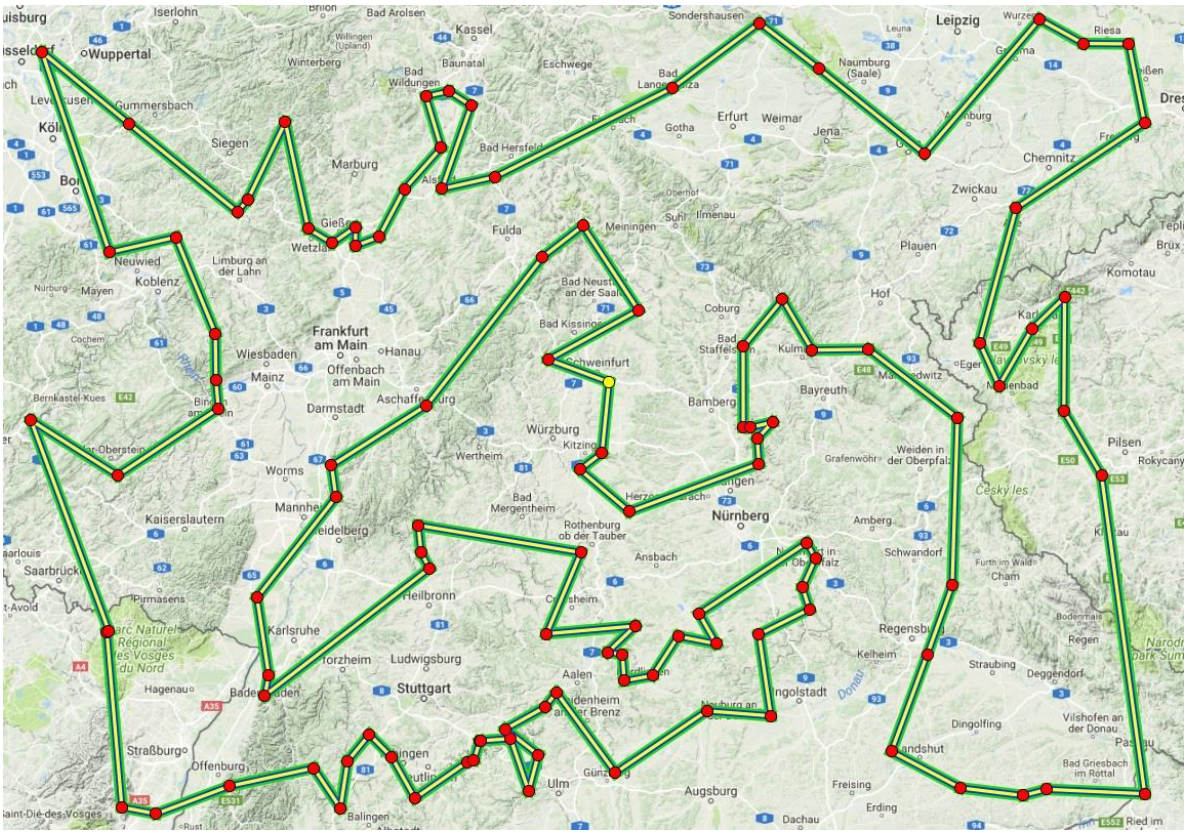


Abbildung 5.5 in QGIS Visualisiertes Ergebnis, mit den drei Layern für CPU(gelb), MCPU(grün) und GPU (blau) übereinandergelegt.

Bei den Visualisierungen ist sehr gut zusehen, dass alle drei Berechnungsarten zu demselben Ergebnis führen. Dies ist für den Vergleich der Berechnungszeiten sehr wichtig, um eine genaue Aussage treffen zu können. Aufgrund der kleinen Instanzgröße, ist in diesem Fall die Berechnungszeit auf der GPU deutlich höher als auf der CPU.

Nach dem in diesem Kapitel ein Exkurs in die Visualisierung mit QGIS stattfand, wird in dem folgenden Kapitel die Arbeit abschließend zusammengefasst.

6. Zusammenfassung

Ziel dieser Thesis war es, ein Programm zum Vergleichen der seriellen und parallelen Berechnungszeiten einer Savings-Heuristik zu schreiben. Das Hauptaugenmerk lag auf parallelen Berechnungen auf der Grafikkarte. Für realitätsnahe Daten wurden die GPS Koordinaten von Flughäfen aus der Datenbank von Openstreetmap.org genutzt. Zur Umsetzung wurde die Programmiersprache C, unter der Verwendung der Programmierschnittstelle CUDA® von NVIDIA, openMP und Microsoft Visual Studio 2013, verwendet.

Am Anfang wurden Grundlagen geschaffen. Hier wurde auf den detaillierten Aufbau moderner Prozessoren sowie Grafikkarten eingegangen. Es wurde beim Transistor begonnen und einzelne Schaltungen dargestellt, mit deren Hilfe ein moderner Computer Berechnungen durchführt. Anschließend wurde in Kapitel 3 die Programmierschnittstelle CUDA® vorgestellt und auf deren Besonderheiten und Verwendung eingegangen. In Kapitel 4 rückte die angewandte Berechnung, die Savings-Heuristik von Clarke und Wright, in den Vordergrund. Als erstes wurde die Heuristik erörtert und im Anschluss auf mögliche Parallelisierungen eingegangen. Das vorletzte Kapitel „Praktischer Teil“ beschäftigte sich mit den speziellen Vorgehensweisen bei der Umsetzung des Programmes. Hier wurde beschrieben, warum welche Technik angewandt wurde.

Die Logistik ist ein sehr weitreichender Bereich. Es gibt sehr viele Lösungsmöglichkeiten für verschiedenste Szenarien. Unzählige Algorithmen, die entwickelt wurden, um mit deren Hilfe Probleme zu lösen, wobei die Digitalisierung nicht aus dem Blick gelassen werden sollte. Während vor rund 70 Jahren der Zeitaufwand für eine hohe Anzahl an Berechnungen sehr groß war, ist dieses mit modernen Computern oftmals in einem Bruchteil von Sekunden möglich. Auch wenn die Entwicklung neuer Algorithmen langsam voranschreitet, steigt der Zuwachs der Geschwindigkeiten von Computern stark an. Allerdings wurde im Zuge dieser Thesis festgestellt, dass die Nutzung von parallelen Berechnungen und nicht genutztes Potenzial der Grafikkarten in der Logistik oftmals nicht berücksichtigt wird.

Abschließend kann gesagt werden, dass die Verwendung dieser Technik durchaus bei einer sehr hohen Anzahl an Berechnungen, zur deutlichen Beschleunigung der Berechnungszeit führt. Dabei kann die Berechnungszeit, je nach Leistung der CPU und GPU, sowie Art der Implementierung, auf der Grafikkarte sehr stark reduziert werden. Zukünftig sollte die Verwendung paralleler Berechnungen auf der Grafikkarte, über die Generierung von Kryptowährung hinaus, in anderen Bereichen berücksichtigt werden.

Literaturverzeichnis

Thomas Friedli & Günther Schuh „Wettbewerbsfähigkeit der Produktion an Hochlohnstandorten“

EN 60617-2 „Referenzhandbuch der international genormten Symbole für elektrische Schaltpläne“

Jürgen Wolf „C von A bis Z“, Galileo Computing

Otto Zinke & Heinrich Brunswig „Lehrbuch der Hochfrequenztechnik“

AIDA64 Tool	https://www.aida64.com/
NVIDIA	http://www.nvidia.de/
CUDA Toolkit Documentation	http://docs.nvidia.com/cuda/
Radixsort	https://en.wikipedia.org/wiki/Radix_sort
Odd Even Sort	https://en.wikipedia.org/wiki/Odd%E2%80%93even_sort
OpenMP	http://www.openmp.org/
GNU Compiler Collection	https://gcc.gnu.org/
OpenMP	https://www.openmp.org/

Abkürzungsverzeichnis

ALU	Arithmetic Logic Unit
CPU	Central Processing Unit
CUDA	Compute United Device Architecture
GPS	Global Positioning System
GPU	Graphics Processing Unit
HTT	Hyper-Threading-Technik
MCPU	Multi Central Processing Unit
QGIS	Quantum Geographisches Informations System
SLI	Scalable Link Interface

Abbildungsverzeichnis

Das Bildmaterial wurde mit Hilfe von Microsoft Office Produkten sowie mit yEd der Firma yWorks GmbH erstellt.

Tabellenverzeichnis